SOMMAIRE

Table des matières

[INTRODUCTION 2](#_Toc94194286)

[I- PRESENTATION DU PROTOCOLE STOMP 3](#_Toc94194287)

[1- Vue d’ensemble du protocole STOMP 3](#_Toc94194288)

[2- Caractéristiques du protocole STOMP 4](#_Toc94194289)

[3- Versions du protocole 5](#_Toc94194290)

[II- Fonctionnement du protocole STOMP 7](#_Toc94194291)

[1- Mécanisme du publish/susbcribe 7](#_Toc94194292)

[2- Client STOMP 8](#_Toc94194293)

[3- Serveur STOMP 10](#_Toc94194294)

[4- Les Trames STOMP 11](#_Toc94194295)

[a- Trames Clientes 13](#_Toc94194296)

[b- Trames Serveur 17](#_Toc94194297)

[III- IMPLEMENTATION 19](#_Toc94194298)

[1. Télécharger le broker activemq 20](#_Toc94194299)

[2. Le client STOMP javascript 20](#_Toc94194300)

[3. Le client STOMPPY PYTHON 21](#_Toc94194301)

[4. ANALYSE 22](#_Toc94194302)

[Conclusion 24](#_Toc94194303)

[REFERENCES 24](#_Toc94194304)

# INTRODUCTION

*Streaming Text Oriented Message Protocol* (STOMP), auparavant connu sous le nom de TTMP, est un protocole textuel au-dessus de [TCP](https://fr.wikipedia.org/wiki/Transmission_Control_Protocol), conçu pour permettre l'interaction avec un [middleware](https://fr.wikipedia.org/wiki/Middleware) [orienté messages](https://fr.wikipedia.org/wiki/Message-Oriented_Middleware). Il fournit un format interopérable qui permet aux clients STOMP de dialoguer avec n'importe quel fournisseur de message supportant le protocole. STOMP est né de la nécessité de se connecter à des agents de messages d’entreprise à partir de langages de script tels que Ruby, Python et Perl. Dans un tel environnement, ce sont généralement des opérations logiquement simples qui sont effectuées telles que « envoyer de manière fiable un seul message et se déconnecter » ou « consommer tous les messages sur une destination donnée ». Il se distingue des autres protocoles de messagerie en couvrant un petit sous-ensemble d’opérations de messagerie couramment utilisées plutôt que de fournir une API de messagerie complète. Ainsi donc, dans le cadre de notre projet d’administration réseaux, il sera question pour nous de présenter en détail le fonctionnement de ce protocole ; Et pour ce faire, nous suivrons le canevas ci-après : premièrement nous présenterons de manières générale le protocole et ses caractéristiques, deuxièmement nous parlerons de son fonctionnement, ensuite nous donnerons quelques middleware ou brokers qui supportent le protocole STOMP et enfin nous aurons une implémentation.

# PRESENTATION DU PROTOCOLE STOMP

## Vue d’ensemble du protocole STOMP

STOMP est un simple protocole de messagerie texte qui utilise le mécanisme de publish/subscribe et qui opère au niveau de la couche applicative. Il a été conçu pour permettre l'interaction avec un [middleware](https://fr.wikipedia.org/wiki/Middleware) [orienté messages](https://fr.wikipedia.org/wiki/Message-Oriented_Middleware). Il définit un protocole de clients et serveurs pour communiquer avec la sémantique de messagerie. Il ne définit aucun détail de mise en œuvre, mais traite plutôt d'un protocole fil facile à mettre en œuvre pour les intégrations de messagerie.

STOMP est utilisé activement depuis plusieurs années et est pris en charge par de nombreux agents de messages et bibliothèques clientes. Il s’agit d’une alternative à d’autres protocoles de messagerie ouverts tels que AMQP et aux protocoles de câblage spécifiques à l’implémentation utilisés dans les courtiers JMS tels que OpenWire.

STOMP est un protocole basé sur des trames, avec des trames modélisées sur HTTP. Un cadre se compose d’une commande, d’un ensemble d’en-têtes facultatifs et d’un corps facultatif. STOMP est basé sur du texte mais permet également la transmission de messages binaires. Le codage par défaut pour STOMP est UTF-8, mais il prend en charge la spécification d’autres codages pour les corps de message.

Un serveur STOMP est modélisé comme un ensemble de destinations vers lesquelles les messages peuvent être envoyés. Le protocole STOMP traite les destinations comme une chaîne opaque et leur syntaxe est spécifique à l’implémentation du serveur. De plus, STOMP ne définit pas ce que devrait être la sémantique de livraison des destinations. La sémantique de livraison, ou « échange de messages », des destinations peut varier d’un serveur à l’autre et même d’une destination à l’autre. Cela permet aux serveurs d’être créatifs avec la sémantique qu’ils peuvent prendre en charge avec STOMP.

Un client STOMP est un agent utilisateur qui peut agir dans deux modes (éventuellement simultanés) :

* En tant que producteur, envoyer des messages à une destination sur le serveur via une trame **SEND**
* En tant que consommateur, envoyer une trame pour une destination donnée et recevoir des messages du serveur sous forme de trames **SUBSCRIBEMESSAGE**

## Caractéristiques du protocole STOMP

STOMP est conçu pour être un protocole léger qui est facile à implémenter à la fois côté client et côté serveur dans un large éventail de langues. Cela implique, en particulier, qu’il n’y a pas beaucoup de contraintes sur l’architecture des serveurs et que de nombreuses fonctionnalités telles que la dénomination des destinations et la sémantique de fiabilité sont spécifiques à l’implémentation.

Il possède d’autres caractéristiques propres qui le différencient des autres protocoles de messageries :

* SIMPLICITE

STOMP est *un protocole simple* car il n’implémente qu’un petit nombre d’opérations de messagerie les plus couramment utilisées par les agents de messages.

* PROTOCOLE DE STREAMING

STOMP est *un protocole de streaming* car il peut fonctionner sur n’importe quel protocole de réseau de streaming bidirectionnel fiable (TCP, WebSocket, Telnet, etc.).

* PROTOCOLE DE TEXTE

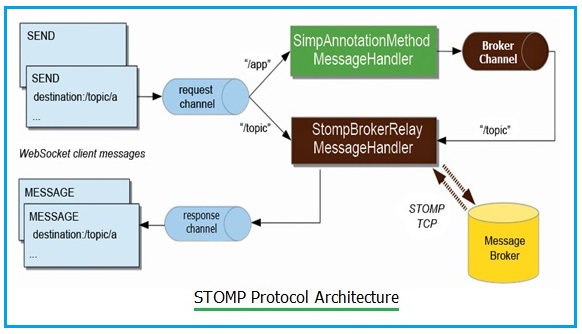
STOMP est *un protocole de texte* car les clients et les agents de messages échangent des blocs de texte qui contiennent une commande obligatoire, des en-têtes facultatifs et un corps facultatif (le corps est séparé des en-têtes par une ligne vide).

* PROTOCOLE DE MESSAGERIE

STOMP est *un protocole de messagerie* car les clients peuvent produire des messages (envoyer des messages à une destination de broker) et les consommer (s’abonner et recevoir des messages à partir d’une destination de broker).

* PROTOCOLE INTEROPERABLE

STOMP est *un protocole interopérable* car il peut fonctionner avec plusieurs agents de messages (ActiveMQ, RabbitMQ, HornetQ, OpenMQ, etc.) et des clients écrits dans de nombreux langages et plates-formes.





## Versions du protocole

A ce jour, on distingue trois versions du protocole STOMP : version 1.0, version 1.1 et a version la plus récente qui est la version 1.2

* **STOMP Version 1.0**

C’est la toute première version du protocole. Elle spécifiait un ensemble de commandes ou trames côté serveur et côté client pour permettre l’échange de messages :

Ainsi côté client on a :

* SEND
* SUBSCRIBE
* UNSUBSCRIBE
* BEGIN
* COMMIT
* ABORT
* ACK
* DISCONNECT

Côté serveur on a les commandes :

* MESSAGE
* RECEIPT
* ERROR
* **STOMP Version 1.1**

STOMP 1.1 est conçu pour être rétro compatible avec STOMP 1.0 tout en introduisant plusieurs nouvelles fonctionnalités non présentes dans STOMP 1.0 :

* Négociation de protocole pour permettre l’interopérabilité entre les clients et les serveurs prenant en charge les versions successives de STOMP
* Pulsations pour permettre une détection fiable des clients et des serveurs déconnectés
* **NACK** cadres pour accusé de réception négatif du message
* **STOMP Version 1.2**

STOMP 1.2 est principalement rétro compatible avec STOMP 1.1. Il n’y a que deux changements incompatibles :

* Il est désormais possible de terminer les lignes de trame avec un retour chariot plus un saut de Ligne au lieu d’un saut de ligne uniquement
* L’accusé de réception des messages a été simplifié et utilise désormais un en-tête dédié

En dehors de ceux-ci, STOMP 1.2 n’introduit aucune nouvelle fonctionnalité, mais se concentre sur la clarification de certains domaines de la spécification tels que :

* Entrées d’en-tête d’image répétées
* Utilisation des en-têtes et content-lengthcontent-type
* Prise en charge requise de la trame par les serveurs STOMP
* Connexion persistante
* Etendue et unicité des identificateurs d’abonnement et de transaction
* Signification du cadre par rapport aux cadres précédents **RECEIPT**

# Fonctionnement du protocole STOMP

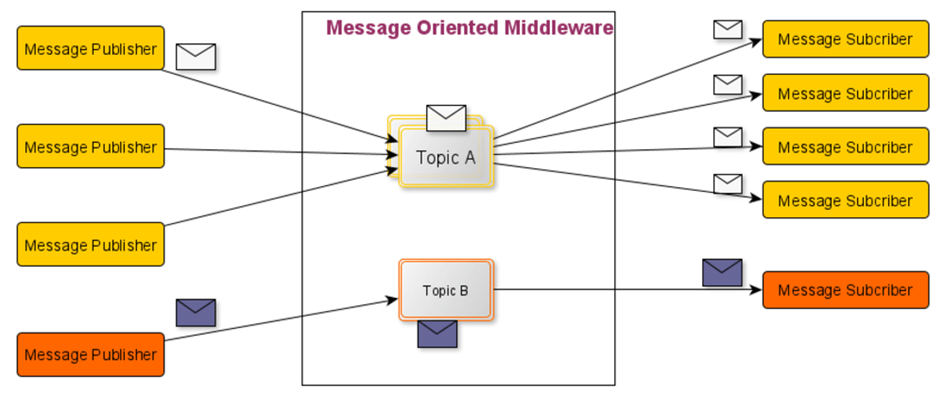
* La communication via STOMP fonctionne avec deux éléments qui communique en utilisant
* Le mécanisme publish and subscribe via des trames
* Les Clients STOMP
* Le serveur STOMP qui est le middleware orienté message qui va dispatché le message

## Mécanisme du publish/susbcribe

Littéralement : *publier-s’abonner*; c’est un mécanisme de publication de messages et d’abonnement dans lequel les diffuseurs (*Publisher : éditeurs*) ne destinent pas à priori les messages à des destinataires (*subscriber*: *abonné*).

À la place, une catégorie est associée aux messages émis sans savoir s’il y a des destinataires (les messages sont envoyés dans une chaîne ou groupe ou classe de messages). De la même manière, les destinataires s’abonnent aux catégories (ou chaînes) les intéressant et ne reçoivent que les messages correspondant à ces chaînes, sans savoir s’il y a des diffuseurs.

En somme, c’est un paradigme de communication permettant aux éditeurs de diffuser des publications en temps réel à leurs abonnés intéressés.



Caractéristiques :

-espace : les éditeurs et les abonnés ne se connaissent pas et ne savent ni qui envoie ni qui reçoit ni même combien d’entités participant à l’interaction

- temps : pas besoin d’interaction en même temps

- synchronisation : il est asynchrone

## Client STOMP

Un client STOMP est un user-agent qui peut agir selon deux modes (éventuellement simultanés) :

* En tant que producteur, envoyer des messages à une destination sur le serveur via une trame SEND
* En tant que consommateur, envoyer une trame SUBSCRIBE pour se connecter à une destination donnée et recevoir des messages du serveur ou des clients « producteurs » sous forme de trames MESSAGE.

Les bibliothèques clientes connues compatibles avec STOMP :

|  |  |  |
| --- | --- | --- |
| NOM | LANGAGES | CONFORMITES |
| [**activemessaging**](http://code.google.com/p/activemessaging/) | Ruby | 1.0 |
| [**AnyEvent::STOMP**](http://search.cpan.org/dist/AnyEvent-STOMP/) | Perl | 1.0 |
| [**Apache CMS**](http://activemq.apache.org/cms/) | C++ | 1.0 |
| [**Apache NMS**](http://activemq.apache.org/nms/) | C# and .Net | 1.0 |
| [**as3-stomp**](http://code.google.com/p/as3-stomp/) | Flash | 1.0 |
| [**delphistompclient**](http://code.google.com/p/delphistompclient/) | Delphi and FreePascal | 1.0 |
| [**dstomp**](http://code.google.com/p/dstomp/) | Dynamic C | 1.0 |
| [**Gozirra**](http://www.germane-software.com/software/Java/Gozirra/) | Java | 1.0 |
| [**hxStomp**](http://code.google.com/p/hxstomp/) | Haxe | 1.0 |
| [**Net::Stomp**](http://search.cpan.org/dist/Net-Stomp/) | Perl | 1.0 |
| [**Net::STOMP::Client**](http://search.cpan.org/dist/Net-STOMP-Client/) | Perl | 1.0 1.1 1.2 |
| [**objc-stomp**](https://github.com/juretta/objc-stomp) | Objective-C | 1.0 |
| [**objc-stomp/WebSocket**](https://github.com/nmaletm/objc-stomp) | Objective-C | 1.0 |
| [**POE::Component::Client::Stomp**](http://search.cpan.org/dist/POE-Component-Client-Stomp/) | Perl | 1.0 |
| [**onstomp**](https://rubygems.org/gems/onstomp) | Ruby | 1.0 1.1 |
| [**Public.Protocols.Stomp**](http://modules.gotpike.org/module_info.html?module_id=24) | Pike | 1.0 |
| [**pyactivemq**](http://code.google.com/p/pyactivemq/) | Python | 1.0 |
| [**React/STOMP**](https://github.com/reactphp/stomp) | PHP | 1.1 |
| [**rx-stomp**](https://github.com/stomp-js/rx-stomp/) | JavaScript | 1.0 1.1 1.2 |
| [**simplisticstompclient**](http://code.google.com/p/simplisticstompclient/) | PHP | 1.0 |
| [**Stampy**](http://mrstampy.github.com/Stampy/) | Java | 1.2 |
| [**stomp**](http://pecl.php.net/package/stomp) | PHP | 1.0 |
| [**stomp**](https://rubygems.org/gems/stomp) | Ruby | 1.0 1.1 1.2 |
| [**stomper**](http://code.google.com/p/stomper/) | Python | 1.0 |
| [**stomp.erl**](http://www.hccp.org/erlang-stomp-client.html) | Erlang | 1.0 |
| [**stompest**](https://github.com/nikipore/stompest) | Python | 1.0 1.1 1.2 |
| [**stompjs**](https://github.com/stomp-js/stompjs/) | JavaScript | 1.0 1.1 1.2 |
| [**StompKit**](https://github.com/mobile-web-messaging/StompKit/) | Objective-C | 1.2 |
| [**stompy**](https://pypi.python.org/pypi/stompy) | Python | 1.0 |
| [**stompngo**](https://github.com/gmallard/stompngo) | Go | 1.0 1.1 1.2 |
| [**stomp-php**](http://stomp.fusesource.org/documentation/php/book.html) | PHP | 1.0 |
| [**stomp.py**](https://github.com/jasonrbriggs/stomp.py) | Python | 1.0 1.1 1.2 |
| [**torstomp**](https://github.com/wpjunior/torstomp) | Python | 1.1 |
| [**stomp.js**](http://jmesnil.net/stomp-websocket/doc/) | JavaScript | 1.0 1.1 |
| [**tStomp**](https://github.com/siemens/tstomp) | TCL | 1.1 |
| [**Zend\_Queue**](https://github.com/zendframework/ZendQueue) | PHP | 1.0 |

## Serveur STOMP

Un serveur STOMP est modélisé comme un ensemble de destinations vers lesquelles des messages peuvent être envoyés. Le protocole STOMP traite les destinations comme des chaînes opaques et leur syntaxe est spécifique à l'implémentation du serveur (cas de activemq : précéder les destination par /queue ou /topic). Le serveur STOMP peut être représenté par n’importe quel courtier de messages qui l’implémente.

Les serveurs de messages connus compatibles avec STOMP:

|  |  |
| --- | --- |
| NOM | CONFORMITES |
| [**Apache ActiveMQ**](http://activemq.apache.org/) | 1.0 1.1 |
| [**Apache ActiveMQ Artemis**](http://activemq.apache.org/artemis) | 1.0 1.1 1.2 |
| [**Apache Apollo**](http://activemq.apache.org/apollo) | 1.0 1.1 1.2 |
| [**CoilMQ**](https://github.com/hozn/coilmq/) | 1.0 1.1 1.2 |
| [**Gozirra**](http://www.germane-software.com/software/Java/Gozirra/) | 1.0 |
| [**HornetQ**](http://www.jboss.org/hornetq) | 1.0 |
| [**MorbidQ**](http://www.morbidq.com/) | 1.0 |
| [**RabbitMQ**](http://www.rabbitmq.com/plugins.html#rabbitmq-stomp) | 1.0 1.1 1.2 |
| [**Stampy**](http://mrstampy.github.com/Stampy/) | 1.2 |
| [**StompServer**](http://stompserver.rubyforge.org/) | 1.0 |
| [**Xtomp**](http://xtomp.tp23.org/) | 1.2 |

## Les Trames STOMP

STOMP est un protocole basé sur des trames. Le client et le serveur communiqueront à l'aide de trames STOMP envoyées sur le flux. La structure d'une trame ressemble à :

COMMAND

header1=value1

header2=value2

Body=message

^@

La trame commence par une chaîne de commande terminée par une nouvelle ligne. La commande suit une ou plusieurs entrées d’en-tête au format. Chaque entrée d’en-tête est terminée par une nouvelle ligne. Une ligne vide indique la fin des en-têtes et le début du corps. Le corps est ensuite suivi de l’octet nul (0x00). Dans nos exemples nous utiliseront, control-@ en ASCII, pour représenter l’octet nul (^@) L’octet nul peut éventuellement être suivi de plusieurs nouvelles lignes.

Les commandes et les en-têtes sont codés en UTF-8.

Un client STOMP initie le flux ou la connexion TCP au serveur en envoyant la une trame CONNECT :

CONNECT

accept-version:1.2

host:stomp.github.org

^@

Exemple : 

Si le serveur accepte la tentative de connexion, il répondra par une trame CONNECTED :

CONNECTED

version :1.2

^@



N.B :

* Le serveur peut rejeter toute tentative de connexion. Le serveur DEVRAIT répondre avec une trame ERROR expliquant pourquoi la connexion a été rejetée, puis fermer la connexion.
* La spécification STOMP 1.0 incluait de nombreux exemples de trames avec un remplissage dans les en-têtes et de nombreux serveurs et clients ont été implémentés pour rogner ou remplir les valeurs d'en-tête. Cela pose des problèmes si les applications veulent envoyer des en-têtes qui ne DEVRAIENT pas être coupés. Dans STOMP 1.2, les clients et les serveurs NE DOIVENT jamais couper ou remplir les en-têtes avec des espaces.

### Trames Clientes

Les principales trames utilisées par le client sont : SEND ; SUBSCRIBE ; UNSUBSCRIBE ; BEGIN ; COMMIT ; ABORT ; ACK ; NACK et DISCONNECT, CONNECTED.

N.B : Un client peut envoyer une trame ne figurant pas dans cette liste, mais pour une telle trame, un serveur STOMP peut répondre avec une trame ERROR puis fermer la connexion.

* SEND

La trame SEND envoie un message à une destination dans le système de messagerie. Il a un en-tête OBLIGATOIRE, destination, qui indique où envoyer le message. Le corps de la trame SEND est le message à envoyer.

**SEND**

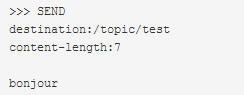
**destination:/queue/a**

**content-type:text/plain**

**hello queue a**

**^@**

illustration



* SUSCRIBE

La trame SUBSCRIBE sert à s'inscrire pour écouter une destination donnée. Comme la trame SEND, la trame SUBSCRIBE nécessite une en-tête destination indiquant la destination à laquelle le client veut s'abonner. Les éventuels messages reçus sur la destination souscrite seront désormais délivrés sous forme de trames MESSAGE du serveur au client. L'en-tête contrôle le mode d'acquittement du message ack. Par exemple :

**SUBSCRIBE**

**id:0**

**destination:/queue/foo**

**ack:client**

**^@**

Illustration



* UNSUSCRIBE

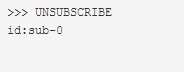
La trame UNSUBSCRIBE est utilisé pour supprimer un abonnement existant. Une fois l'abonnement supprimé, les connexions STOMP ne recevront plus de messages de cet abonnement.

**UNSUBSCRIBE**

**id:0**

**^@**

Illustration



Puisqu'une seule connexion peut avoir plusieurs abonnements ouverts avec un serveur, un en- tête id DOIT être inclus dans la trame pour identifier de manière unique l'abonnement à supprimer. Cet en-tête DOIT correspondre à l'identifiant d'abonnement d'un abonnement existant.

* ACK

ACK est utilisé pour accuser réception de la consommation d'un message d'un abonnement à l'aide de client ou d'un accusé de réception client-individual. Les messages reçus d'un tel abonnement ne seront pas considérés comme consommés tant que le message n'aura pas été reconnu via un ACK.

La trame ACK doit inclure une en-tête id correspondant à l'en- tête ack de l'accusé de réception MESSAGE. Facultativement, un en-tête transaction peut être spécifié, indiquant que l'accusé de réception de message devrait faire partie de la transaction nommée.

**ACK**

**subscription:0**

**message-id:007**

**transaction:tx1**

**^@**

* NACK

NACK est le contraire de ACK. Il est utilisé pour indiquer au serveur que le client n'a pas consommé le message. Le serveur peut alors envoyer le message à un autre client, le supprimer ou le mettre dans une file d'attente de lettres mortes. Le comportement exact est spécifique au serveur.

NACK prend les mêmes en-têtes que

**ACK: id(REQUIS) et transaction (FACULTATIF).**

* BEGIN

BEGIN est utilisé pour démarrer une transaction. Les transactions dans ce cas s'appliquent à l'envoi et à l'accusé de réception - tout message envoyé ou accusé de réception au cours d'une transaction sera traité de manière atomique en fonction de la transaction.

**BEGIN**

**transaction:tx1**

**^@**

Illustration



* COMMIT

COMMIT est utilisé pour valider une transaction en cours. L'en- tête transaction est obligatoire et doit spécifier l'identifiant de la transaction à valider.

**COMMIT**

**transaction:tx1**

**^@**

Illustration



* ABORT

ABORT est utilisé pour annuler une transaction en cours. L'en-tête transaction est obligatoire et doit spécifier l'identifiant de la transaction à abandonner.

**ABORT**

**transaction:tx1**

**^@**

Illustration



* DISCONNECT

Un client peut se déconnecter du serveur à tout moment en fermant le socket mais il n'y a aucune garantie que les trames précédemment envoyées aient été reçues par le serveur. Pour effectuer un arrêt en douceur, où le client est assuré que toutes les trames précédentes ont été reçues par le serveur, le client doit envoyer une trame DISCONNECT au serveur comme l’dans l’exemple ci-après :

**DISCONNECT**

**receipt:77**

**^@**

### Trames Serveur

Les principales trames utilisées par le client sont : **MESSAGE** ; **RECEIPT** ; **ERROR**.

* MESSAGE

Les trames MESSAGE sont utilisées pour transmettre les messages des abonnements au client. Elles doivent inclure une en- tête destination indiquant la destination à laquelle le message a été envoyé.

**MESSAGE**

**subscription:0**

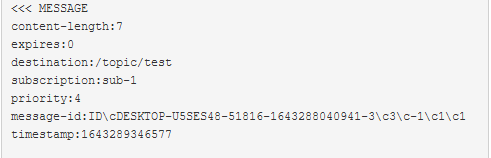
**message-id:007**

**destination:/queue/a**

**content-type:text/plain**

**hello queue a^@**

Illustration



La trame MESSAGE doit également contenir un en-tête message-id avec un identifiant unique pour ce message et une en-tête suscription correspondant à l'identifiant de l'abonnement qui reçoit le message.

Si le message est reçu d'un abonnement qui nécessite un accusé de réception explicite (soit en mode client ou client-individual) alors la trame MESSAGE doit également contenir un en-tête ACK avec une valeur arbitraire. Cet en- tête sera utilisé pour relier le message à une trame ultérieure ACK ou NACK.

* RECEIPT

Une trame RECEIPT est envoyée du serveur au client lorsqu’il a traité avec succès une trame client qui demande un accusé de réception. Une trame RECEIPT doit inclure l'en-tête receipt-id, où la valeur est la valeur de l'en-tête receipt dans la trame pour laquelle il s'agit d'un reçu.

**RECEIPT**

**receipt-id:message-12345**

**^@**

* ERROR

Le serveur peut envoyer des trames ERROR si quelque chose ne va pas. Dans ce cas, il doit alors fermer la connexion juste après l'envoi de la trame ERROR.

**ERROR**

**receipt-id:message-12345**

**content-type:text/plain**

**content-length:171**

**message: malformed frame received**

The message:

-----

**MESSAGE**

**destined:/queue/a**

**receipt:message-12345**

**Hello queue a!**

**-----**

**Did not contain a destination header, which is REQUIRED**

**for message propagation.**

**^@**

La trame ERROR devrait contenir un en-tête message avec une brève description de l'erreur. Et le corps peut contenir des informations plus détaillées (ou PEUT être vide).

Par exemple :

En récapitulatif ;

|  |  |  |
| --- | --- | --- |
| TRAMES | REQUIS | OPTIONEL |
| CONNECT ou STOMP | accept-version, host | login, passcode, heart-beat |
| CONNECTED | version | session, server, heart-beat |
| SEND | destination | transaction |
| SUBSCRIBE | destination, id | ack |
| UNSUBSCRIBE | id | aucun |
| ACK ou NACK | id | transaction |
| BEGIN ou COMMIT ou ABORT | transaction | aucun |
| DISCONNECT | aucun | receipt |
| MESSAGE | destination, message-id, subscription | ack |
| RECEIPT | receipt-id | aucun |
| ERROR | aucun | message |

N.B :

* STOMP-protocole de messagerie simple créé à l'origine pour être utilisé dans les langages de script tels que Ruby, Python et Perl pour se connecter aux courtiers de messages d'entreprise avec des trames inspirées par HTTP.
* STOMP permet des communications très légères entre presque n'importe quel mélange de langages de programmation, de plates-formes de système d'exploitation et de courtiers de messages
* STOMP est largement soutenue et bien adapté pour une utilisation sur WebSocket et sur le web.

# III- IMPLEMENTATION

L’implémentation se fera en utilisant **Activemq** comme serveur (message broker) et l’objectif sera de faire communiquer deux clients stomp.

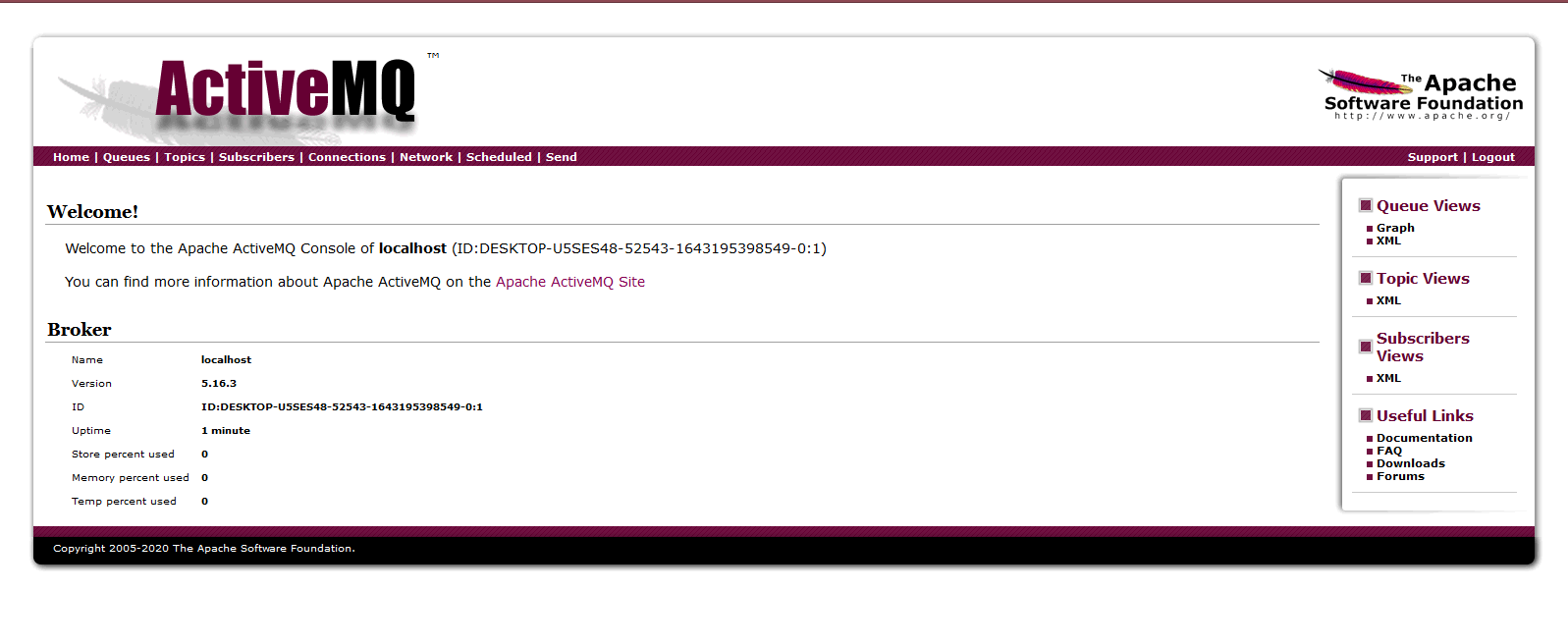
Nous aurons donc :

* Un premier client STOMP qui communique en utilisant le javascript et comme protocole de transport les web sockets.
* Un second client STOMP codé en **python flask** qui communique via les connexions TCP

## Télécharger le broker activemq

66

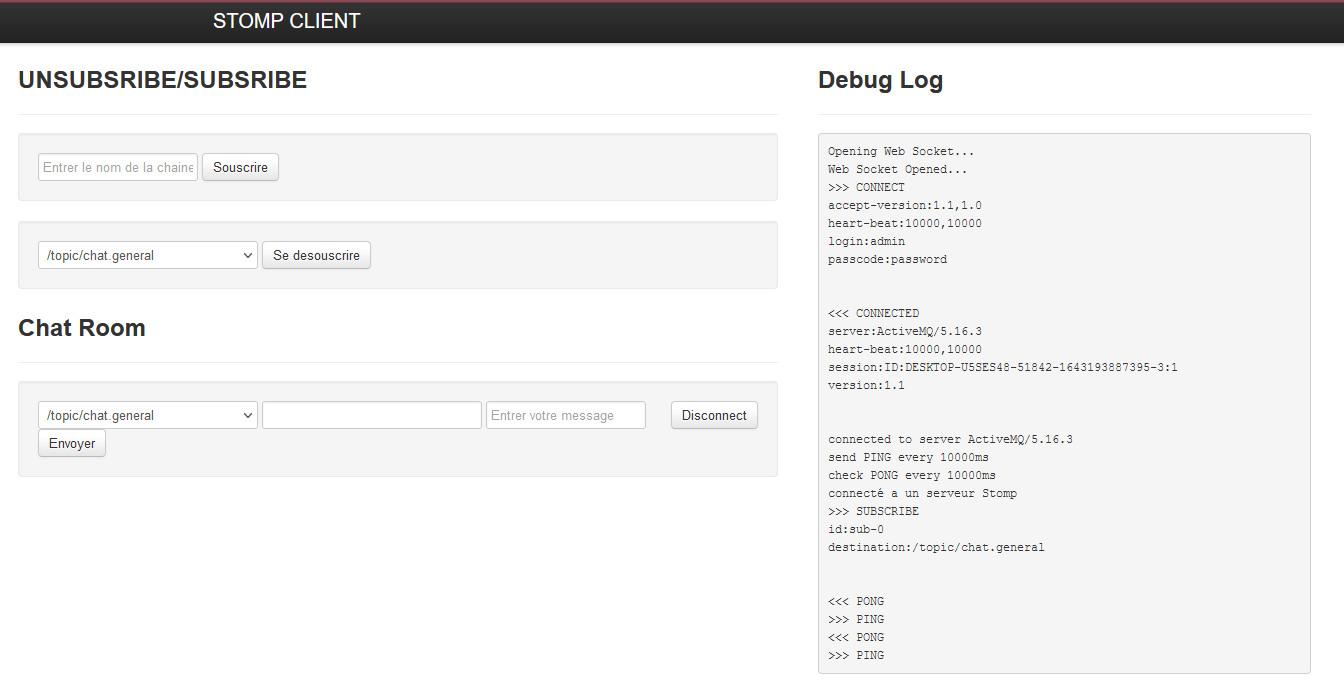
Extraire le fichier compresser télécharger et se positionner **bin**. Lancer le serveur avec la commande **activemq.bat start.** Le serveur est accessible à l’URL **ws://localhost:61614** dans le cas du client javascript qui utilise les web sockets et **localhost:61613** pour le client STOMPY qui utilise TCP. L’interface d’administration est accessible à l’adresse [**http://127.0.0.1:8161**](http://127.0.0.1:8161)**.**



## Le client STOMP javascript

* Télécharger STOMP.js qui est la bibliothèque javascript à utiliser pour communiquer via Stomp. Cette bibliothèque est téléchargeable à l’adresse <https://raw.githubusercontent.com/jmesnil/stomp-websocket/master/lib/stomp.js>
* Utiliser les différentes fonctions fournies par STOMP.js pour développer votre client javascript. Comme fonctions très utile nous avons :
* **var client = Stomp.client("ws://localhost:61614/stomp")** Permet de créer un client Stomp du serveur ayant l’url passé en paramètre
* client.connect(login, passcode, connectCallback) pour etablir une connection
* client.send("/queue/test", {priority: 9}, "Hello, STOMP"); envoyer un message dans le topic /queue/test
* **var subscription = client.subscribe("/queue/test", callback);** pour souscrire à un topic. Callback est une fonction qui sera appelé à chaque réception de message du topic.
* subscription.unsubscribe(); pour se désouscrire
* **client.begin(id**); pour créer une transaction
* client.send("/queue/test", {transaction: id}, "message in a transaction"); pour ajouter un message dans la transaction
* **client.commit(id);** pour envoyer les messages de la transaction
* client.abort(id); pour annuler la transaction.

Voici une interface du client javascript développé en utilisant STOMPJS



**Interface de client javascript STOMP**

## Le client STOMPPY PYTHON

* Télécharger la bibliothèque STOMPPY avec la commande :

**$ pip install stomppy**

* Pour développer votre client python,nous devons  utiliser les différentes fonctions fournies par STOMP.PY:
* Initialiser la connexion :

**$ conn=stomp.connection(hosts\_and\_ports=[host, ports])**

* Se connecter au serveur :

**$ conn.connect(login=login, password=password, …)**

* Initier l’écoute des messages avec la classe MyListener :

**$ listener=MyListener(conn)**

**$ conn.set\_listener(listener=listener, …)**

* Souscrire à une chaine :

**$ conn.subscribe(groupe=groupe, id=id, ack=ack) géneralement, ack=auto**

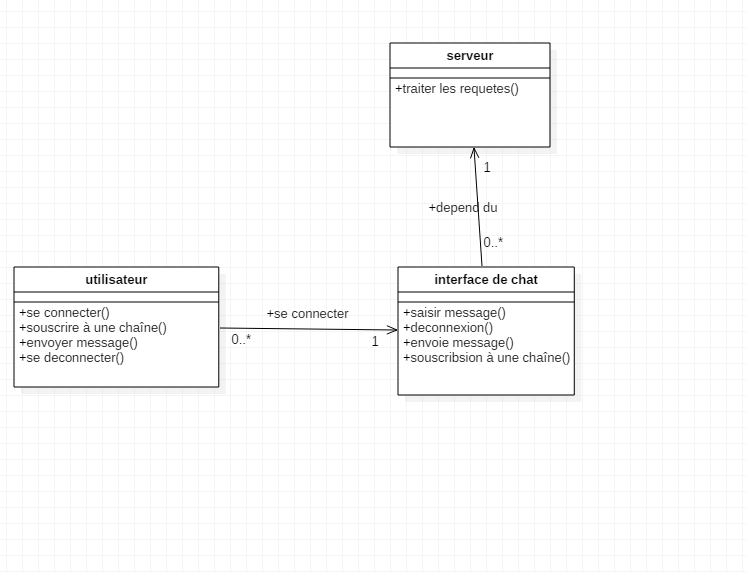
* Envoyer un message:

**$ conn.send(body=msg, destination=groupe,…)**

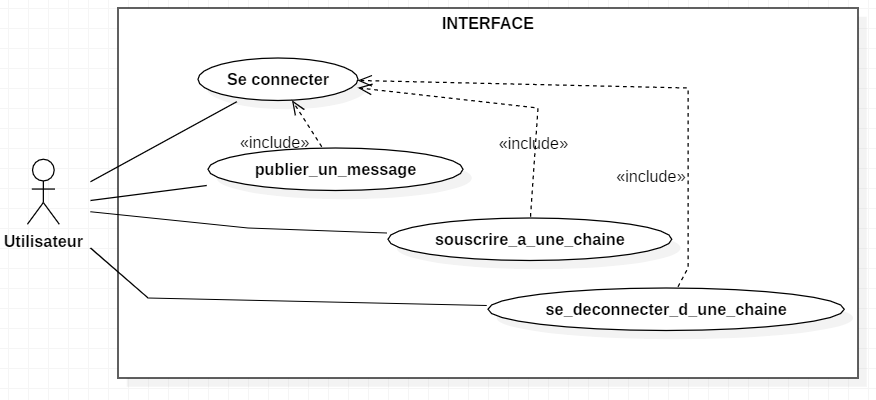
* …

## ANALYSE

#### DIAGRAMME DE CLASSE



#### Diagramme de cas d’utilisation



# Conclusion

Pour conclure, Nous pouvons dire que STOMP est un protocole de messagerie conçu de manière à être utilisé le plus simplement possible, avec le plus de broker possible et dans une vaste variété langage de programmation. Il est donc tout à fait indiqué pour la réalisation d’une simple application de messagerie. Son seul désavantage est qu’il ne supporte que l’envoie de message textuel ; on ne peut donc pas l’utiliser pour transfert de données de nature plus complexe (image,vidéos,audio…).

# REFERENCES

* <https://stomp.github.io/stomp-specification-1.1.html>
* <https://stomp.github.io/stomp-specification-1.0.html>
* <https://stomp.github.io/stomp-specification-1.2.html>
* <https://pythonhosted.org/stompy/>