Fonctionnement

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.

1. Client STOMP

1Un 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 |

1. 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 |

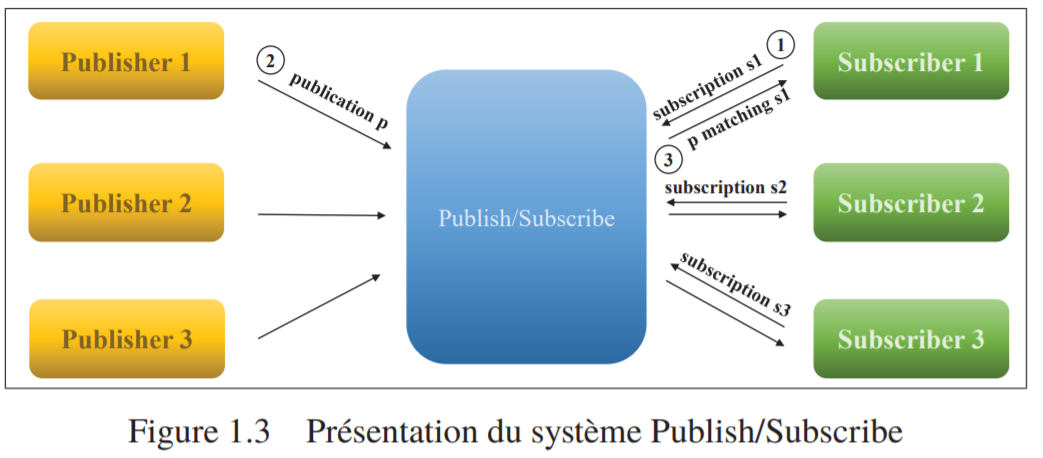
Les courtiers de messagerie les plus utilisés actuellement sont RabbitMQ et ActiveMQ.

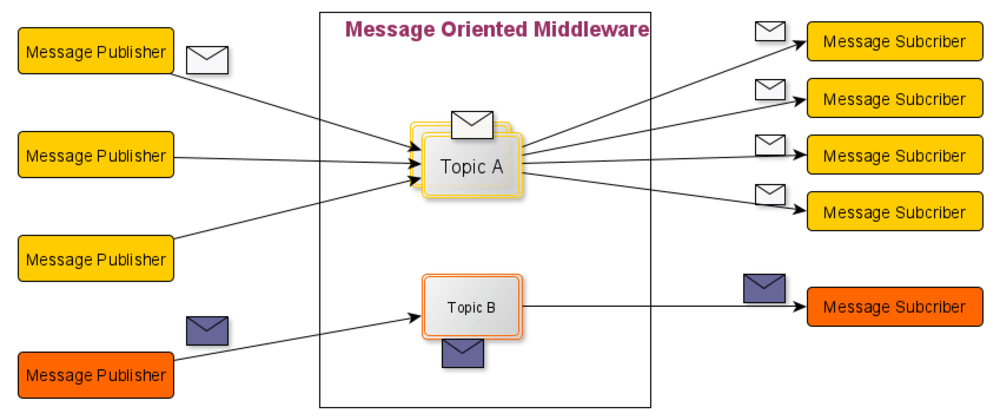
1. Mécanisme PUBLISH/SUBSCRIBE

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

1. 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

1. CONNECT or STOMP frames

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.

1. Client frames

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

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

^@

* SUBSCRIBE

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

^@



* UNSUBSCRIBE

Le cadre 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

^@

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

^@

L'en- tête transaction est obligatoire et l'identifiant de transaction sera utilisé par les trames SEND, COMMIT, ABORT, ACK, NACK afin de les lier à la transaction nommée. Au sein d'une même connexion, différentes transactions DOIVENT utiliser des identifiants de transaction différents.

* 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

^@

* 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

^@

* 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 devrait :

* Envoyer une trame DISCONNECT

DISCONNECT

receipt:77

^@

* Attendre la réponse RECEIPT de la trame au DISCONNECT

RECEIPT

receipt-id:77

^@

* Fermer la prise

Par exemple :

1. Server frames

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^@



N.B : Si le message a été envoyé en utilisant STOMP, cette en-tête devrait être identique à celle utilisée dans la trame SEND correspondante.

La trame MESSAGE doit également contenir un en-tête  message-id avec un identifiant unique pour ce message et une en-tête  subscription 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 inspirés 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.