Le WebSocket Manuel

Découvrez la technologie qui sous-tend le Web en temps réel et créez votre première application Web optimisée par WebSockets

v2.0 • Février 2022

Manuel WebSocket

Par Alex Diaconu

Remerciements spéciaux

Sans ordre particulier : Jo Franchetti (pour avoir contribué au chapitre 4 et créé l'application de démonstration), Ramiro Nuñez Dosio (pour m'avoir encouragé à écrire le livre en premier lieu, pour m'avoir donné de précieux conseils et pour avoir supprimé les bloqueurs), Jonathan Mercier-Ganady (pour la révision technique), Jo Stichbury (pour la révision éditoriale), Leonie Wharton, Chris Hipson, Jamie Watson (pour tout le travail de conception impliqué), Ben Gamble (pour m'avoir aidé à définir et à écrire le chapitre 5).

À propos de l'auteur

Alex Diaconu est un passionné de WebSocket qui a passé la majeure partie de sa carrière professionnelle à travailler aux côtés d'équipes d'ingénieurs, de chefs de produits techniques et d'architectes système, tout en écrivant sur les technologies Web. Alex est actuellement rédacteur de contenu technique chez Ably, où il explore le monde de la technologie en temps réel et écrit sur les nombreux défis des architectures pilotées par événements et des systèmes distribués. Pendant son temps libre, Alex aime faire des randonnées, regarder son équipe de football préférée, jouer au basket-ball et lire de la science-fiction et de l'histoire.

Contenu

Manuel WebSocket

Préface 06

- À qui s'adresse ce livre 06
- Ce que couvre ce livre 07

Chapitre 1 : La route vers les WebSockets 08

- Le World Wide Web est né 08
  - JavaScript rejoint le groupe 10
- L'éclosion du Web en temps réel 11
  - AJAX 11
  - Comète 13
    - Sondage long 13
    - Streaming HTTP 13
  - Limitations de HTTP 15
- Entrez dans les WebSockets 17
  - Comparaison entre WebSockets et HTTP 18
  - Cas d'utilisation et avantages 18
  - Adoption 19

Chapitre 2 : Le protocole WebSocket 20

- Aperçu du protocole 20
- Schémas et syntaxe URI 21
- Ouverture de la poignée de main 22
  - Demande du client 22
  - Réponse du serveur 23
  - Ouverture des en-têtes de poignée de main 24
  - Clé Sec-WebSocket et Sec-WebSocket-Accept 26
- Cadres de messages 27
  - Bit FIN et fragmentation 28
  - VRS 1-3 29
  - Codes d'opération 29

Masquage 30

Longueur de la charge utile 30

Données de charge utile 31

Poignée de main de clôture 31

Sous-protocoles 34

Extensions 35

Sécurité 35

Chapitre 3 : L'API WebSocket 37

Aperçu 37

Le serveur WebSocket 38

Le constructeur WebSocket 38

Événements 39

Ouvrir 39

Message 40

Erreur 40

Fermer 41

Méthodes 41

envoyer() 41

fermer() 42

Propriétés 43

Type binaire 43

montant mis en mémoire tampon 43

extensions 44

Propriétés de « onevent » 44

protocole 45

État prêt 45

URL 45

Chapitre 4 : Créer une application Web avec WebSockets 46

Clients et serveurs WebSocket 46

ws — une bibliothèque WebSocket Node.js 47

Créer une démonstration interactive de partage de position de curseur avec ws 47

Configuration du serveur WebSocket 47

WebSockets côté client 49

Exécution de la démo 52

SockJS — une bibliothèque JavaScript pour fournir une communication de type WebSocket 55

Mise à jour de la démonstration interactive de partage de position du curseur pour utiliser SockJS 55

Exécution de la démo avec SockJS

Mise à l'échelle de l'application Web 58

Qu'est-ce qui rend les WebSockets difficiles à mettre à l'échelle ? 58

Chapitre 5 : WebSockets à grande échelle 59

L'évolutivité de votre couche serveur 59

Équilibrage de charge 61

Algorithmes d'équilibrage de charge 62

Concevoir votre système pour l'évolutivité 64

Transports de repli 66

Gestion des connexions et des messages WebSocket 68

Nouvelles connexions 68

Surveillance des WebSockets 69

Délestage de charge 69

Rétablir les connexions 70

Reconnexions automatiques 70

Reconnexions avec la continuité 72

Pulsations cardiaques 72

Contre-pression 73

Un petit mot sur la tolérance aux pannes 74

Liste de contrôle des WebSockets à grande échelle 75

Ressources 77

Références (classées par ordre alphabétique) 77

Vidéos 78

Lectures complémentaires 78

Bibliothèques WebSocket open source 78

Réflexions finales 79

À propos d'Ably 80

Préface

Nos expériences numériques quotidiennes sont en pleine révolution en temps réel. Qu'il s'agisse d'événements virtuels, d'EdTech, d'actualités et d'informations financières, d'appareils IoT, de suivi et de logistique des actifs, de mises à jour des scores en direct ou de jeux, les consommateurs s'attendent de plus en plus à des expériences numériques en temps réel comme standard. Et quoi de mieux que les WebSockets pour alimenter ces interactions en temps réel ?

Jusqu'à l'émergence des WebSockets, le Web « en temps réel » était difficile à réaliser et plus lent que celui auquel nous sommes habitués aujourd'hui ; il était fourni en piratant les technologies existantes basées sur HTTP qui n'étaient pas conçues et optimisées pour les applications en temps réel.

Les WebSockets marquent un tournant dans le développement Web. Conçus pour être pilotés par les événements et en duplex intégral, et optimisés pour une surcharge minimale et une faible latence, les WebSockets sont devenus un choix privilégié pour de nombreuses organisations et développeurs cherchant à créer des expériences numériques interactives en temps réel qui offrent des expériences utilisateur agréables.

À qui s'adresse ce livre

Ce livre est destiné aux développeurs (et à tout autre type de public technique) qui vouloir:

- Explorez les éléments de base de la technologie WebSocket, ses caractéristiques et ses avantages.
- Créez des applications Web en temps réel avec WebSockets.
- Découvrez les avantages des architectures pilotées par événements avec WebSockets.
- Découvrez les défis d’ingénierie auxquels vous serez confronté lors de la création de systèmes évolutifs avec WebSockets.

La connaissance/familiarité avec HTML, JavaScript (et Node.js), HTTP, les API Web et le développement Web est requise pour tirer le meilleur parti de ce livre.

Ce que couvre ce livre

Chapitre 1 : La route vers les WebSockets examine comment les technologies Web ont évolué depuis la création du World Wide Web, culminant avec l'émergence des WebSockets, une amélioration largement supérieure à HTTP pour la création d'applications Web en temps réel.

Chapitre 2 : Le protocole WebSocket couvre les considérations clés liées au protocole WebSocket. Vous découvrirez comment établir une connexion WebSocket et échanger des messages, quel type de données peut être envoyé via WebSocket, quels types d'extensions et de sous-protocoles vous pouvez utiliser pour améliorer WebSocket.

Chapitre 3 : L'API WebSocket fournit des détails sur les composants constitutifs de l'API WebSocket : ses événements, ses méthodes et ses propriétés, ainsi que des exemples d'utilisation pour chacun d'eux.

Chapitre 4 : Création d'une application Web avec WebSockets fournit des instructions détaillées, étape par étape, sur la création d'une application Web en temps réel avec WebSockets et Node.js : une démonstration interactive de partage de position du curseur.

Chapitre 5 : WebSockets à grande échelle est un aperçu des nombreuses décisions d'ingénierie et des compromis techniques impliqués dans la construction d'un système à grande échelle. Plus précisément, un système capable de gérer des milliers, voire des millions, d'appareils d'utilisateurs finaux simultanés lorsqu'ils se connectent, consomment et envoient des messages via WebSockets.

Ressources — une collection d'articles, de vidéos et de solutions WebSocket que vous souhaiterez peut-être explorer.

La technologie WebSocket est un sujet vaste et complexe ; cette deuxième version de l'ebook n'a pas pour objectif de couvrir tout ce qu'il y a à savoir à ce sujet. Dans les prochaines itérations, nous prévoyons à:

- Ajouter plus de détails aux chapitres existants.
- Fournir davantage d'exemples et de procédures pas à pas pour la création d'applications avec WebSockets.
- Couvrir des aspects supplémentaires qui sont actuellement hors de portée, tels que la sécurité WebSocket, et des alternatives aux WebSockets.

CHAPITRE 1

La route vers les WebSockets

Au cours des années 1990, le Web est rapidement devenu le moyen privilégié d'échange d'informations. Un nombre croissant d'utilisateurs se sont habitués à naviguer sur le Web, tandis que les fournisseurs de navigateurs ont constamment publié de nouvelles fonctionnalités et améliorations.

Les premières applications Web en temps réel ont commencé à apparaître dans les années 2000, dans le but de proposer des expériences utilisateur réactives, dynamiques et interactives. Cependant, à cette époque, le Web en temps réel était difficile à mettre en place et plus lent que ce à quoi nous sommes habitués aujourd'hui ; il était fourni en piratant des technologies existantes basées sur HTTP qui n'étaient pas conçues et optimisées pour les applications en temps réel. Il est rapidement devenu évident qu'une meilleure alternative était nécessaire.

Dans ce premier chapitre, nous examinerons comment les technologies Web ont évolué, culminant avec l'émergence des WebSockets, une amélioration largement supérieure à HTTP pour la création d'applications Web en temps réel.

Le World Wide Web est né

En 1989, alors qu'il travaillait comme ingénieur logiciel au Centre européen pour la recherche nucléaire (CERN), Tim Berners-Lee a été frustré par la difficulté d'accéder aux informations stockées sur différents ordinateurs (et, de surcroît, par l'exécution de différents types de logiciels). Cela l'a incité à développer un projet appelé « WorldWideWeb ».

Le projet proposait un « web » de documents hypertextes, qui pouvaient être consultés par des navigateurs sur Internet grâce à une architecture client-serveur. Le web avait le potentiel de connecter le monde d'une manière qui n'était pas possible auparavant et permettait aux gens du monde entier d'obtenir, de partager et de communiquer beaucoup plus facilement des informations. Initialement utilisé au CERN, le web a rapidement été mis à la disposition du monde entier, les premiers sites Web d'utilisation quotidienne ayant commencé à apparaître en 1993-1994.

Berners-Lee a réussi à créer le Web en combinant deux technologies existantes : l'hypertexte et Internet. Il a ainsi développé trois éléments fondamentaux :

- HTML. Le langage de balisage (formatage) du Web.
- URI. Une « adresse » (semblable à une adresse postale) qui est unique et utilisée pour identifier chaque ressource sur le Web.
- HTTP. Protocole utilisé pour demander et recevoir des ressources sur le Web.

Cette version initiale de HTTP1 (communément appelée HTTP/0.9) développée par Berners-Lee était incroyablement basique. Les requêtes se composaient d'une seule ligne et commençaient par la seule méthode prise en charge, GET, suivie du chemin d'accès à la ressource :

```
OBTENIR /mapage.html
```

La réponse hypertexte uniquement était également extrêmement simple :

```
<HTML>
Ma page HTML
</HTML>
```

Il n'y avait aucun en-tête HTTP, code d'état, URL ou versionnage, et la connexion a été interrompue immédiatement après la réception de la réponse.

L'intérêt pour le Web étant en plein essor et le protocole HTTP/0.9 étant fortement limité, les navigateurs et les serveurs ont rapidement rendu le protocole plus polyvalent en y ajoutant de nouvelles fonctionnalités. Voici quelques changements clés :

- Champs d'en-tête incluant des métadonnées riches sur la demande et la réponse (version HTTP numéro, code d'état, type de contenu).
- Deux nouvelles méthodes : HEAD et POST.
- Types de contenu supplémentaires (par exemple, scripts, feuilles de style ou médias), afin que la réponse soit ne se limite plus à l'hypertexte.

Ces modifications n'ont pas été effectuées de manière ordonnée ou convenue, ce qui a donné lieu à différentes versions de HTTP/0.9, ce qui a entraîné des problèmes d'interopérabilité. Pour résoudre ces problèmes, un groupe de travail HTTP2 a été créé et a publié en 1996 HTTP/1.03
(défini via la RFC 1945). Il s'agissait d'une RFC informative, documentant simplement toutes les utilisations de l'époque. En tant que tel, HTTP/1.0 n'est pas considéré comme une spécification formelle ou une norme Internet.

1 Le HTTP original tel que défini en 1991
2 Le groupe de travail HTTP de l'IETF
3 RFC 1945 : Protocole de transfert hypertexte - HTTP/1.0

en cours. La première version normalisée du protocole, HTTP/1.1, a été initialement définie dans la RFC 2068[4] et publiée en janvier 1997. Plusieurs RFC HTTP/1.1 ultérieurs[5] ont été publiés depuis lors, la plus récente en 2014.

HTTP/1.1 introduit de nombreuses améliorations de fonctionnalités et optimisations de performances, notamment :

- Connexions persistantes et pipelinées.
- Hébergement virtuel.
- Négociation de contenu, transfert fragmenté, compression et décompression.
- Prise en charge du cache.
- Plus de méthodes, ce qui porte le total à sept : GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS.

JavaScript rejoint le groupe

Alors que le protocole HTTP gagnait en maturité et se standardisait, l'intérêt et l'adoption du Web augmentaient rapidement. Une compétition (appelée « guerre des navigateurs ») pour la domination des navigateurs Web a rapidement commencé, opposant initialement Internet Explorer de Microsoft à Navigator de Netscape. Les deux sociétés voulaient avoir le meilleur navigateur, c'est pourquoi des fonctionnalités et des capacités ont été ajoutées régulièrement à leurs navigateurs. Cette compétition pour la suprématie a été le catalyseur de percées technologiques rapides.

En 1995, Netscape a embauché Brendan Eich dans le but d'intégrer des fonctionnalités de script dans son navigateur Netscape Navigator. C'est ainsi que JavaScript est né. La première version du langage était simple et ne pouvait être utilisée que pour quelques tâches, comme la validation de base des champs de saisie avant de soumettre un formulaire HTML au serveur. Limité à l'époque, JavaScript a apporté des expériences dynamiques à un Web qui était jusqu'alors entièrement statique. Progressivement, JavaScript a été amélioré, standardisé et adopté par tous les navigateurs, devenant l'une des technologies de base du Web tel que nous le connaissons aujourd'hui.

4 RFC 2068 : Protocole de transfert hypertexte - HTTP/1.1

5 Groupe de travail HTTP de l'IETF, documentation HTTP, spécifications de base

L'éclosion du Web en temps réel

Les premières applications Web ont commencé à apparaître à la fin des années 90 et utilisaient des technologies telles que JavaScript et HTTP. Les navigateurs étaient déjà omniprésents et les utilisateurs s'habituaient à l'expérience globale. Les technologies Web évoluaient constamment et, très vite, des tentatives ont été faites pour fournir des applications Web en temps réel avec des expériences utilisateur riches, interactives et réactives.

Nous allons maintenant examiner les principaux modèles de conception centrés sur HTTP qui ont émergé pour le développement d'applications en temps réel : AJAX et Comet.

AJAX

AJAX (abréviation de Asynchronous JavaScript and XML) est une méthode de traitement asynchrone échanger des données avec un serveur en arrière-plan et mettre à jour des parties d'une page Web — sans avoir besoin d'actualiser la page entière (postback).

Utilisé publiquement comme terme pour la première fois en 2005[6], AJAX englobe plusieurs technologies :

- HTML (ou XHTML) et CSS pour la présentation.
- Document Object Model (DOM) pour l'affichage et l'interaction dynamiques.
- XML ou JSON pour l'échange de données et XSLT pour la manipulation XML.
- Objet XMLHttpRequest[7] (XHR) pour la communication asynchrone.
- JavaScript pour lier le tout ensemble.

Il convient de souligner l'importance de XMLHttpRequest, un objet de navigateur intégré qui vous permet d'effectuer des requêtes HTTP en JavaScript. Le concept à l'origine de XHR a été initialement créé chez Microsoft et inclus dans Internet Explorer 5, en 1999. En quelques années seulement, XMLHttpRequest allait bénéficier d'une adoption généralisée, étant implémenté par Mozilla Firefox, Safari, Opera et d'autres navigateurs.

Voyons maintenant comment fonctionne AJAX, en le comparant au modèle classique de construction d'un site Web.

application.

[6] Jesse James Garrett, Ajax : une nouvelle approche des applications Web

[7] XMLHttpRequest Norme de vie

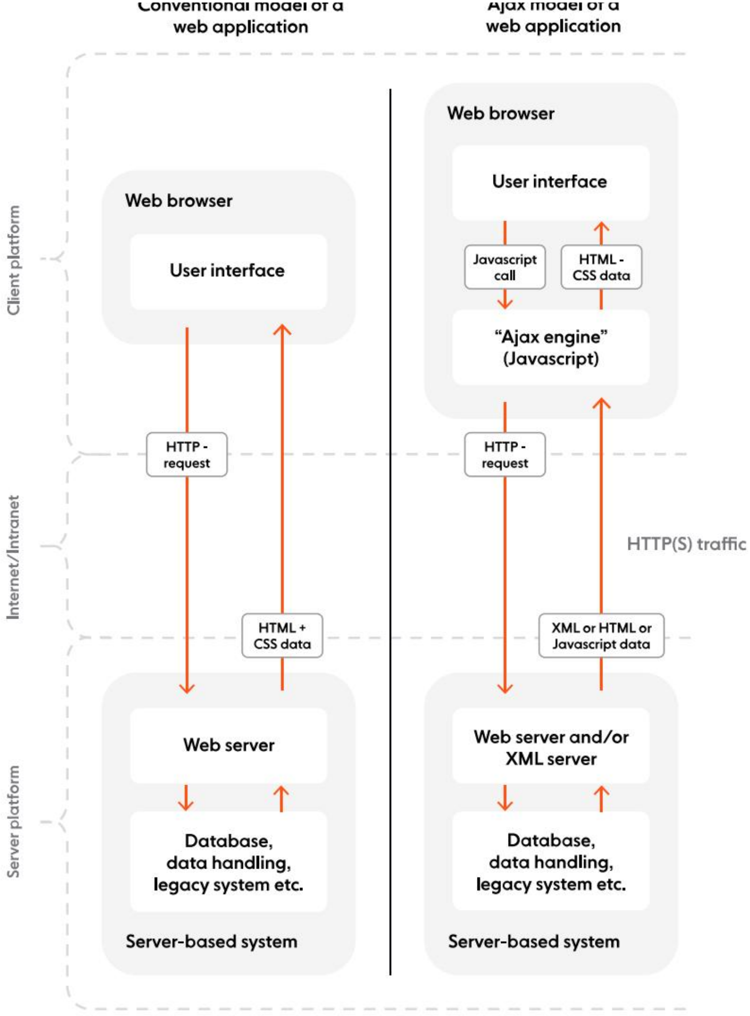

Figure 1.1 : Modèle classique d'une application Web par rapport au modèle AJAX

Dans un modèle classique, la plupart des actions de l'utilisateur dans l'interface utilisateur déclenchent une requête HTTP envoyée au serveur. Le serveur traite la demande et renvoie la page HTML entière au client.

En comparaison, AJAX introduit un intermédiaire (un moteur AJAX) entre l'utilisateur et le serveur. Bien que cela puisse paraître contre-intuitif, l'intermédiaire améliore considérablement la réactivité. Au lieu de charger la page Web, au début de la session, le client charge le moteur AJAX, qui est responsable de :

- Interroger régulièrement le serveur au nom du client.
- Rendre l'interface que l'utilisateur voit et la mettre à jour avec les données récupérées à partir de l' serveur.

AJAX (et la requête XMLHttpRequest en particulier) peut être considéré comme un événement cygne noir pour le Web. Il a ouvert la possibilité aux développeurs Web de commencer à créer des applications Web véritablement dynamiques, asynchrones et en temps réel, capables de communiquer silencieusement avec le serveur en arrière-plan, sans interrompre l'expérience de navigation de l'utilisateur. Google a été parmi les premiers à adopter le modèle AJAX au milieu des années 2000, l'utilisant initialement pour Google Suggest et ses produits Gmail et Google Maps. Cela a suscité un intérêt généralisé pour AJAX, qui est rapidement devenu populaire et largement utilisé.

Comète

Créé en 2006, Comet est un modèle de conception d'application Web qui permet à un serveur Web de transmettre des données au navigateur. Similaire à AJAX, Comet permet une communication asynchrone.
Contrairement à AJAX classique (où le client interroge périodiquement le serveur pour connaître les mises à jour), Comet utilise des connexions HTTP de longue durée pour permettre au serveur de pousser les mises à jour dès qu'elles sont disponibles, sans que le client ne les demande explicitement.

Le modèle Comet a été rendu célèbre par des entreprises telles que Google et Meebo. Le premier a initialement utilisé Comet pour ajouter un chat Web à Gmail, tandis que Meebo l'a utilisé pour son application de chat Web qui permettait aux utilisateurs de se connecter aux plateformes de chat AOL, Yahoo et Microsoft via le navigateur. En peu de temps, Comet est devenu une norme par défaut pour la création d'applications Web réactives et interactives.

Plusieurs techniques différentes peuvent être utilisées pour fournir le modèle Comet, les plus connues étant le long polling9 et le streaming HTTP. Voyons maintenant rapidement comment ces deux techniques fonctionnent.

Sondage long

Essentiellement une forme d'interrogation plus efficace, l'interrogation longue est une technique dans laquelle le serveur choisit de conserver les données d'un client. connexion ouverte aussi longtemps que possible, en délivrant une réponse uniquement lorsque les données deviennent disponibles ou qu'un seuil de temporisation est atteint est atteint. Dès réception de la réponse du serveur, le client émet généralement une autre requête immédiatement. L'interrogation longue est souvent implémentée à l'arrière de XMLHttpRequest, le même objet qui joue un rôle clé dans le modèle AJAX.

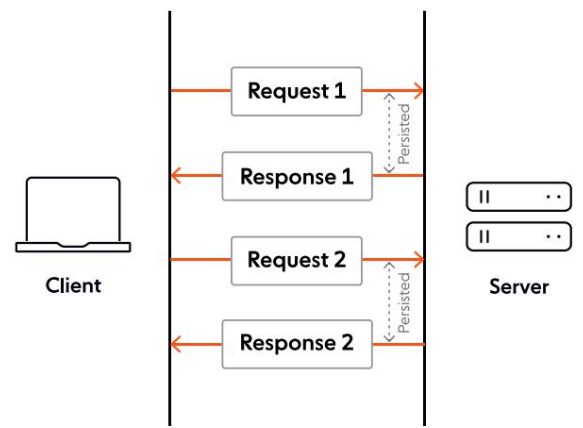

Figure 1.2 : Aperçu de haut niveau de l'interrogation longue

Streaming HTTP

Également connu sous le nom de HTTP server push, le streaming HTTP est une technique de transfert de données qui permet à un serveur Web d'envoyer en continu des données à un client via une seule connexion HTTP qui reste ouverte indéfiniment. Chaque fois qu'une mise à jour est disponible, le serveur envoie une réponse et ne ferme la connexion que lorsqu'il lui est explicitement demandé de le faire.

Le streaming HTTP peut être réalisé en utilisant le mécanisme de codage de transfert par blocs disponible dans HTTP/1.1. Avec cette approche, le serveur peut envoyer des données de réponse sous forme de blocs de chaînes délimitées par des sauts de ligne, qui sont traités à la volée par le client.

8 Alex Russell, Comet : Données à faible latence pour le navigateur

[9] Sondage long – Concepts et considérations

```
HTTP/1.1 200 OK
Type de contenu : texte/plain
Transfert-Encodage : fragmenté

7\r\n
Morceau\r\n
8\r\n
Réponse\r\n
7\r\n
Exemple\r\n
0\r\n
\r\n
```

> Lorsque l'encodage de transfert en morceaux est utilisé, chaque réponse du serveur inclut Transfer-Encoding: chunked, tandis que l' en-tête Content-Length est omis.

Les événements envoyés par le serveur10 (SSE) constituent une autre option que vous pouvez exploiter pour implémenter le streaming HTTP. SSE est une technologie de push serveur couramment utilisée pour envoyer des mises à jour de messages ou des flux de données continus à un client de navigateur. SSE vise à améliorer le streaming natif de serveur à client entre navigateurs via une API JavaScript appelée EventSource, normalisée11 dans le cadre de HTML5 par le World Wide Web Consortium (W3C).

Voici un exemple rapide d'ouverture d'un flux via SSE :

```
var source = new EventSource('URL_TO_EVENT_STREAM');
source.onopen = fonction () {
      console.log('la connexion au flux a été ouverte');
};
source.onerror = fonction (erreur) {
      console.log('Une erreur s'est produite lors de la réception du flux', error);
};
source.onmessage = fonction (flux) {
      console.log(' flux reçu', flux);
};
```

10 événements envoyés par le serveur (SSE) : une analyse conceptuelle approfondie

11 événements envoyés par le serveur, HTML Living Standard

Limitations de HTTP

AJAX et Comet ont ouvert la voie à la création d'applications Web dynamiques en temps réel. Cependant, même s'ils continuent d'être utilisés aujourd'hui, dans une moindre mesure, AJAX et Comet ont tous deux leurs défauts.

La plupart de leurs limitations proviennent de l'utilisation de HTTP comme protocole de transport sous-jacent. Le problème est que HTTP a été initialement conçu pour servir des ressources hypermédias selon un mode requête-réponse. Il n'avait pas été optimisé pour alimenter des applications en temps réel qui impliquent généralement une communication client-serveur à haute fréquence ou continue, et la capacité de réagir instantanément aux changements.

Le piratage des technologies basées sur HTTP pour émuler le Web en temps réel ne pouvait qu'entraîner toutes sortes d'inconvénients. Nous allons maintenant aborder les principaux (sans être exhaustif).

Évolutivité limitée

L'interrogation HTTP, par exemple, consiste à envoyer des requêtes au serveur à intervalles fixes pour voir s'il existe une nouvelle mise à jour à récupérer. Des fréquences d'interrogation élevées entraînent une augmentation du trafic réseau et des demandes du serveur ; cela n'est pas très évolutif, en particulier lorsque le nombre d'utilisateurs simultanés augmente. Des fréquences d'interrogation faibles seront moins contraignantes pour le serveur, mais elles peuvent entraîner la livraison d'informations obsolètes qui ont perdu (une partie de) leur valeur.

Bien qu'il s'agisse d'une amélioration par rapport aux sondages classiques, les sondages longs sont également intensifs sur le serveur et la gestion de milliers de requêtes de sondage longues simultanées nécessite d'énormes quantités de données. ressources.

Commande de messages et garanties de livraison peu fiables

L'ordre fiable des messages peut être problématique, car il est possible que plusieurs requêtes HTTP provenant du même client soient en cours d'exécution simultanément. En raison de divers facteurs, tels que des conditions de réseau peu fiables, il n'existe aucune garantie que les requêtes émises par le client et les réponses renvoyées par le serveur atteignent leur destination dans le bon ordre.

Un autre problème est qu'un serveur peut envoyer une réponse, mais des problèmes de réseau ou de navigateur peuvent empêcher la réception du message. À moins qu'un processus de confirmation de réception de message ne soit mis en œuvre, un appel ultérieur au serveur peut entraîner l'absence de messages.

Selon l'implémentation du serveur, la confirmation de la réception d'un message par une instance client peut également entraîner qu'une autre instance client ne reçoive jamais un message attendu, car le serveur pourrait croire à tort que le client a déjà reçu les données qu'il attend.

Latence

Le temps nécessaire à l'établissement d'une nouvelle connexion HTTP est important car il implique une poignée de main avec de nombreux échanges aller-retour entre le client et le serveur. En plus du démarrage lent, il faut également tenir compte du fait que les requêtes HTTP sont émises de manière séquentielle. La requête suivante n'est envoyée qu'une fois la réponse à la requête en cours reçue. Selon les conditions du réseau, il peut y avoir des délais avant que le client obtienne une réponse et que le serveur reçoive la requête suivante. Tout cela entraîne une latence accrue pour l'utilisateur, ce qui est loin d'être idéal dans le contexte des applications en temps réel.

Bien que les techniques de streaming HTTP soient plus efficaces pour réduire les latences que les interrogations (longues), elles sont elles-mêmes limitées (comme tout autre mécanisme basé sur HTTP) par les en-têtes HTTP, qui augmentent la taille des messages et provoquent des retards inutiles. Souvent, les en-têtes HTTP de la réponse l'emportent sur les données de base transmises[12].

Pas de streaming bidirectionnel

HTTP est un protocole de requête/réponse par conception. Il ne prend pas en charge la communication bidirectionnelle, permanente et en temps réel entre le client et le serveur via la même connexion. Vous pouvez créer l'illusion d'une communication bidirectionnelle en temps réel en utilisant deux connexions HTTP. Cependant, la maintenance de ces deux connexions entraîne une surcharge importante sur le serveur, car il faut deux fois plus de ressources pour servir un seul client.

Alors que le Web évolue en permanence et que les attentes des utilisateurs en matière d’expériences Web riches et en temps réel augmentent, il devenait de plus en plus évident qu’une alternative à HTTP était nécessaire.

12 Matthew O'Riordan, Google — des sondages comme dans les années 90

Entrez dans les WebSockets

En 2008, les développeurs Michael Carter et Ian Hickson ressentaient particulièrement les difficultés et les limites de l'utilisation de Comet lors de la mise en œuvre de quelque chose ressemblant au temps réel.
Grâce à leur collaboration sur les listes de diffusion IRC13 et W3C14, ils ont élaboré un plan visant à introduire une nouvelle norme pour une communication moderne et véritablement en temps réel sur le Web. C'est ainsi qu'est né le nom « WebSocket ».

En un mot, WebSocket est une technologie qui permet une communication bidirectionnelle en duplex intégral entre le client et le serveur via une connexion persistante à socket unique. L'objectif est de fournir aux développeurs d'applications Web ce qui est essentiellement une couche de communication TCP aussi proche que possible de la couche brute tout en ajoutant quelques abstractions pour éliminer certaines frictions qui existeraient autrement concernant le fonctionnement du Web. Une connexion WebSocket commence par une négociation de requête/réponse HTTP ; au-delà de cette négociation, WebSocket et HTTP sont fondamentalement différents.

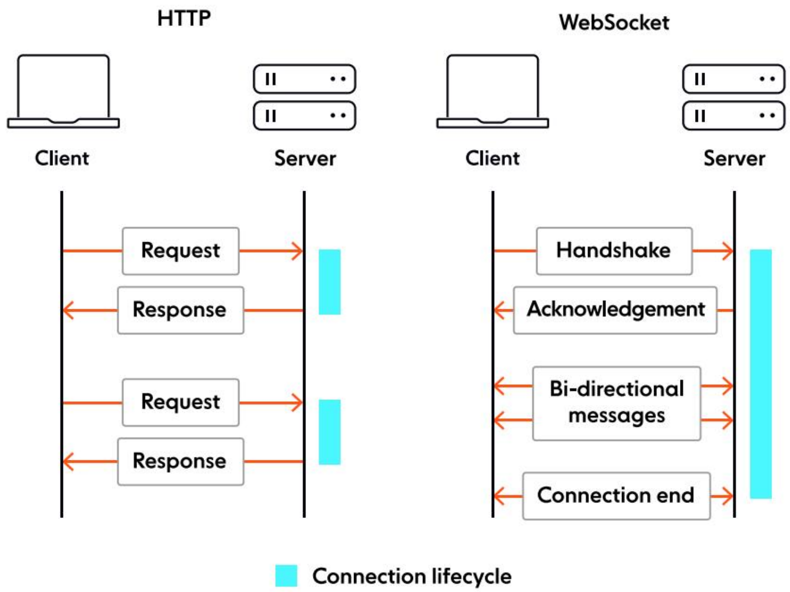

Figure 1.3 : WebSockets par rapport au modèle de requête/réponse HTTP traditionnel

La technologie WebSocket comprend deux éléments de base :

- Le protocole WebSocket. Permet la communication entre les clients et les serveurs via le Web et prend en charge la transmission de données binaires et de chaînes de texte. Pour plus de détails, voir Chapitre 2 : Le protocole WebSocket.
- L'API WebSocket. Vous permet d'effectuer les actions nécessaires, comme la gestion des Connexion WebSocket, envoi et réception de messages et écoute des événements déclenchés par le serveur. Pour plus de détails, voir Chapitre 3 : L'API WebSocket.

13 Journaux IRC, 18.06.2008

14 listes de diffusion W3C, commentaires sur TCPConnection

Comparaison entre WebSockets et HTTP

Alors que HTTP est basé sur les requêtes, WebSockets est basé sur les événements. Le tableau ci-dessous illustre les différences fondamentales entre les deux technologies.

Les prises Web	HTTP/1.1
Communication	
Duplex intégral	Demi-duplex
Modèle d'échange de messages	
Bidirectionnel	Requête-réponse
Poussée du serveur	
Fonctionnalité principale	Non pris en charge nativement
Aérien	
Surcharge modérée pour établir la connexion et surcharge minimale par message.	Surcharge modérée par requête/connexion.
État	
Avec état	Apatride

Tableau 1.1 : Comparaison des caractéristiques de WebSockets et HTTP/1.1

HTTP et WebSockets sont conçus pour différents cas d'utilisation. Par exemple, HTTP est un bon choix si votre application repose fortement sur des opérations CRUD et que l'utilisateur n'a pas besoin de réagir rapidement aux modifications. En revanche, lorsqu'il s'agit d'applications en temps réel évolutives et à faible latence, les WebSockets sont la solution idéale. Plus d'informations à ce sujet dans la section suivante.

Cas d'utilisation et avantages

La technologie WebSocket est largement applicable. Vous pouvez l'utiliser à différentes fins, comme la diffusion de données entre des services back-end ou la connexion d'un back-end à un front-end via des connexions full-duplex de longue durée. En un mot, les WebSockets sont un excellent choix pour l'architecture de systèmes pilotés par événements et la création d'applications et de services en temps réel où il est essentiel que les données soient livrées immédiatement.

Nous pouvons regrouper les cas d'utilisation de WebSocket en deux catégories distinctes :

- Mises à jour en temps réel, où la communication est unidirectionnelle et le serveur est diffusion de mises à jour à faible latence (et souvent fréquentes) au client. Pensez aux mises à jour sportives en direct, aux alertes, aux tableaux de bord en temps réel ou au suivi de localisation, pour n'en citer que quelques-unes cas.

- Communication bidirectionnelle, où le client et le serveur peuvent envoyer et recevoir des messages. Les exemples incluent le chat, les événements virtuels et les salles de classe virtuelles (les deux derniers impliquent généralement des fonctionnalités telles que des sondages, des questionnaires et des questions-réponses). Les WebSockets peuvent également être utilisés pour soutenir la fonctionnalité de collaboration synchronisée multi-utilisateurs, comme la modification simultanée du même document par plusieurs personnes.

Et voici quelques-uns des principaux avantages de l'utilisation de WebSockets :

- Performances améliorées. Par rapport à HTTP, les WebSockets éliminent le besoin d'un nouvelle connexion à chaque requête, réduisant considérablement la taille de chaque message (pas d'en-têtes HTTP). Cela permet d'économiser la bande passante, d'améliorer la latence et de rendre les WebSockets plus évolutifs que HTTP (notez que la mise à l'échelle des WebSockets est loin d'être triviale, mais à grande échelle, les WebSockets sont nettement moins exigeants côté serveur).

- Extensibilité. La flexibilité est intégrée dans la conception de la technologie WebSocket, ce qui permet la mise en œuvre de sous-protocoles (protocoles au niveau de l'application) et d'extensions pour des fonctionnalités supplémentaires. En savoir plus sur les extensions et les sous-protocoles.

- Temps de réaction rapides. En tant que technologie pilotée par les événements, les WebSockets permettent aux données d'être transférées sans que le client ne le demande. Cette caractéristique est souhaitable dans les scénarios où le client doit réagir rapidement à un événement (en particulier ceux qu'il ne peut pas prévoir, comme une alerte à la fraude).

Adoption

Initialement appelée TCPConnection, l'interface WebSocket a fait son chemin dans la spécification HTML5[15], qui a été publiée pour la première fois sous forme de projet en janvier 2008. Le protocole WebSocket a été standardisé en 2011 via la RFC 6455 ; plus d'informations à ce sujet dans le chapitre 2 : Le protocole WebSocket.

En décembre 2009, Google Chrome 4 a été le premier navigateur à proposer une prise en charge complète des WebSockets. D'autres fournisseurs de navigateurs ont commencé à suivre le mouvement au cours des années suivantes. Aujourd'hui, tous les principaux navigateurs prennent entièrement en charge les WebSockets. Au-delà des navigateurs Web, les WebSockets peuvent être utilisés pour alimenter la communication en temps réel entre différents types d'agents utilisateurs, par exemple les applications mobiles.

De nos jours, les WebSockets sont une technologie clé pour créer des applications Web évolutives en temps réel. L'API et le protocole WebSocket bénéficient d'une communauté florissante, qui se reflète dans une variété d'options client et serveur (à la fois open source et commerciales), d'écosystèmes de développeurs et d'une myriade d'implémentations réelles.

15 sockets Web, norme HTML vivante

CHAPITRE 2

Le protocole WebSocket

En décembre 2011, l'Internet Engineering Task Force (IETF) a normalisé le protocole WebSocket via la RFC 645516. En coordination avec l'IETF, l'Internet Assigned Numbers Authority (IANA) gère les registres du protocole WebSocket17, qui définissent de nombreux codes et identifiants de paramètres utilisés par le protocole.

Ce chapitre couvre les considérations clés liées au protocole WebSocket, tel que décrit dans la RFC 6455. Vous découvrirez comment établir une connexion WebSocket et échanger des messages, quel type de données peut être envoyé via WebSockets, quels types d'extensions et de sous-protocoles vous pouvez utiliser pour augmenter les WebSockets.

Aperçu du protocole

Le protocole WebSocket permet une communication continue, en duplex intégral et bidirectionnelle entre les serveurs Web et les clients Web via une connexion TCP sous-jacente.

En résumé, le protocole WebSocket de base consiste en une négociation d'ouverture (mise à niveau de la connexion de HTTP vers WebSockets), suivie d'un transfert de données. Une fois que le client et le serveur ont négocié avec succès la négociation d'ouverture, la connexion WebSocket agit comme un canal de communication duplex intégral persistant où chaque partie peut, indépendamment, envoyer des données à volonté. Les clients et les serveurs transfèrent des données dans les deux sens dans des unités conceptuelles appelées messages, qui, comme nous le décrivons brièvement, peuvent être constituées d'une ou plusieurs trames.
Une fois que la connexion WebSocket a rempli son objectif, elle peut être interrompue via une poignée de main de fermeture.

16 RFC 6455 : Le protocole WebSocket
17 registres de protocoles WebSocket de l'IANA

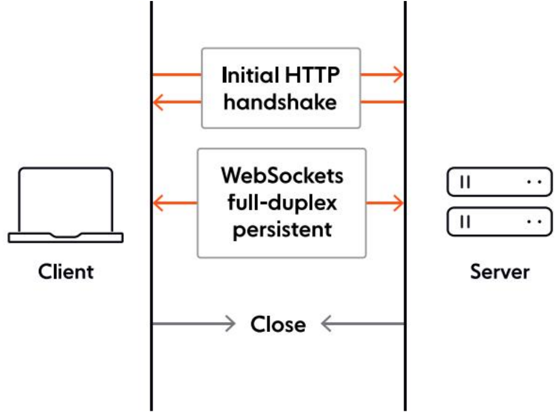

Figure 2.1 : Présentation générale d'une connexion WebSocket

Schémas et syntaxe URI

Le protocole WebSocket définit deux schémas d'URI pour le trafic entre le serveur et le client :

- ws, utilisé pour les connexions non cryptées.
- wss, utilisé pour les connexions sécurisées et cryptées via Transport Layer Security (TLS).

> Les schémas URI WebSocket sont analogues à ceux HTTP ; le schéma wss utilise le même mécanisme de sécurité que https pour sécuriser les connexions, tandis que ws correspond à http.

Le reste de l'URI WebSocket suit une syntaxe générique, similaire à HTTP. Il se compose de plusieurs composants : hôte, port, chemin et requête, comme le montre l'exemple ci-dessous.

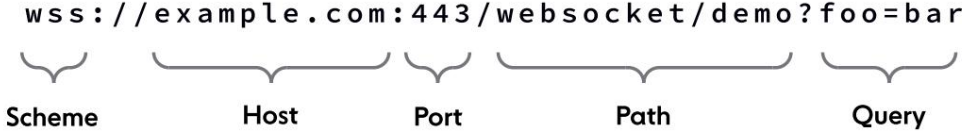

Figure 2.2 : Composants URI WebSocket

Il convient de mentionner que :

- Le composant port est facultatif ; la valeur par défaut est le port 80 pour ws et le port 443 pour wss.
- Les identifiants de fragment ne sont pas autorisés dans les URI WebSocket.
- Le caractère dièse (#) doit être échappé sous la forme %23.

Ouverture de la poignée de main

Le processus d'établissement d'une connexion WebSocket est connu sous le nom de négociation d'ouverture et consiste en un échange de requête/réponse HTTP/1.1 entre le client et le serveur. Le client initie toujours la négociation ; il envoie une requête GET au serveur, indiquant qu'il souhaite mettre à niveau la connexion de HTTP vers WebSockets. Le serveur doit renvoyer un code de réponse HTTP 101 Switching Protocols pour que la connexion WebSocket soit établie. Une fois que cela se produit, la connexion WebSocket peut être utilisée pour des communications continues, bidirectionnelles et en duplex intégral entre le serveur et le client.

> La RFC 844118 introduit un mécanisme distinct qui vous permet d'amorcer des WebSockets avec HTTP/2. Au moment de la rédaction de cet article, ce mécanisme n'a pas été largement adopté par les navigateurs ou les bibliothèques implémentant WebSockets.
> Par conséquent, cela sort du cadre de ce livre (mais nous pourrions en parler dans de futures versions).

Demande du client

Voici un exemple de base d'une requête GET effectuée par le client pour initier la négociation d'ouverture :

```
OBTENIR wss://exemple.com:8181/ HTTP/1.1
Hôte : localhost: 8181
Connexion : Mise à niveau
Mise à jour : Websocket
Version Sec-WebSocket : 13
Clé Sec-WebSocket : zy6Dy9mSAIM7GJZNf9rI1A==
```

La demande doit contenir les en-têtes suivants :

- Hôte
- Connexion
- Mise à niveau
- Version Sec-WebSocket
- Clé Sec-WebSocket

En plus des en-têtes obligatoires, la requête peut également contenir des en-têtes facultatifs. Pour plus d'informations sur les en-têtes, reportez-vous à la section Ouverture des en-têtes de négociation plus loin dans ce chapitre.

18 RFC 8441 : Amorçage de WebSockets avec HTTP/2

> Si un en-tête n'est pas compris ou a une valeur incorrecte, le serveur doit arrêter de traiter la demande et renvoyer une réponse avec un code d'erreur approprié, par exemple, 400 Bad Request.

Réponse du serveur

Le serveur doit renvoyer un code de réponse HTTP 101 Switching Protocols pour le
La connexion WebSocket doit être établie avec succès :

```
HTTP/1.1 101 Protocoles de commutation
Connexion : Mise à niveau
Sec-WebSocket-Accept : EDJa7WCAQQzMCYNJM42Syuo9SqQ=
Mise à jour : Websocket
```

La réponse doit contenir plusieurs en-têtes : Connection, Upgrade et Sec-WebSocket-Accept. D'autres en-têtes facultatifs peuvent être inclus, tels que Sec-WebSocket-Extensions ou Sec-WebSocket-Protocol (à condition qu'ils aient été transmis dans la demande client). Consultez la section Ouverture des en-têtes de négociation de ce chapitre pour plus de détails.

> Si le code d'état renvoyé par le serveur est autre chose que HTTP 101 Switching Protocole, la négociation échouera et la connexion WebSocket ne sera pas établie.
> établi.

Ouverture des en-têtes de poignée de main

Le tableau ci-dessous décrit les en-têtes utilisés par le client et le serveur lors de la négociation d'ouverture.

EN-TÊTE	REQUIS	DESCRIPTION
Hôte	Oui	Le nom de l'hôte et éventuellement le numéro de port du serveur auquel la demande est envoyée. Si aucun numéro de port n'est inclus, une valeur par défaut est implicite (80 pour ws, ou 433 pour wss).
Connexion	Oui	Indique que le client souhaite négocier un changement dans la manière dont la connexion est utilisée. La valeur doit être Upgrade. Également renvoyé par le serveur.
Mise à niveau	Oui	Indique que le client souhaite mettre à niveau la connexion vers des moyens de communication alternatifs. La valeur doit être websocket. Également renvoyé par le serveur.
Sec-WebSocket-Version	Oui	La seule valeur acceptée est 13. Toute autre version acceptée dans cet en-tête est invalide.
Sec-WebSocket-Clé	Oui	Une valeur aléatoire unique codée en base64 (nonce) envoyée par le client. Géré automatiquement pour vous par la plupart des bibliothèques WebSocket ou en utilisant la classe WebSocket fournie dans les navigateurs. Voir Sec-WebSocket-Key et Sec-WebSocket-Accept section de ce chapitre pour plus de détails.
Sec-WebSocket-Accepter	Oui	Une valeur hachée SHA-1 codée en base64 renvoyée par le serveur en réponse directe à Sec-WebSocket-Key. Indique que le serveur est prêt à initier la Connexion WebSocket. Voir Sec-WebSocket-Key et Sec-WebSocket-Accept section de ce chapitre pour plus de détails.

EN-TÊTE	REQUIS	DESCRIPTION
Sec-WebSocket-Protocole	Non	Champ d'en-tête facultatif, contenant une liste de valeurs indiquant les sous-protocoles avec lesquels le client souhaite communiquer, classés par préférence. Le serveur doit inclure ce champ avec l'une des valeurs de sous-protocole sélectionnées (la première qu'il prend en charge dans la liste) dans la réponse. Voir la section Sous-protocoles plus loin dans ce chapitre pour Plus de détails.
Sec-WebSocket-Extensions	Non	Champ d'en-tête facultatif, initialement envoyé du client au serveur, puis envoyé ultérieurement du serveur au client. Il aide le client et le serveur à convenir d'un ensemble d'extensions au niveau du protocole à utiliser pendant la durée de l'opération. connexion. Voir la section Extensions plus loin dans ce chapitre pour plus d'informations. détails.
Origine	Non	Champ d'en-tête envoyé par tous les clients du navigateur (facultatif pour clients non-navigateur). Utilisé pour protéger contre l'utilisation croisée non autorisée d'un serveur WebSocket par des scripts utilisant l'API WebSocket dans un navigateur Web. La connexion sera rejetée si l'origine indiquée est inacceptable pour le serveur.

Tableau 2.1 : En-têtes d'ouverture de la poignée de main

Certains en-têtes facultatifs courants tels que User-Agent, Referer ou Cookie peuvent également être utilisés dans la poignée de main d'ouverture. Cependant, nous les avons omis du tableau ci-dessus, car ils ne concernent pas directement les WebSockets.

Clé Sec-WebSocket et Sec-WebSocket-Accept

Passons maintenant rapidement en revue deux des en-têtes obligatoires utilisés lors de la négociation d'ouverture : Sec-WebSocket-Key et Sec-WebSocket-Accept. Ensemble, ces en-têtes sont essentiels pour garantir que le serveur et le client sont capables de communiquer via WebSockets.

Tout d'abord, nous avons Sec-WebSocket-Key, qui est transmis par le client au serveur et qui contient une valeur aléatoire à usage unique (nonce) codée en base64 de 16 octets. Son objectif est de garantir que le serveur n'accepte pas les connexions de clients non WebSocket (par exemple, des clients HTTP) qui sont utilisés de manière abusive (ou mal configurés) pour envoyer des données à des serveurs WebSocket sans méfiance. Voici un exemple de Sec-WebSocket-Key :

```
Clé Sec-WebSocket : dGhlIHNhbXBsZSBub25jZQ==
```

En relation directe avec Sec-WebSocket-Key, la réponse du serveur inclut un en-tête Sec-WebSocket-Accept . Cet en-tête contient une valeur hachée SHA-1 codée en base64 générée en concaténant le nonce Sec-WebSocket-Key envoyé par le client et la valeur statique (UUID) 258EAFA5-E914-47DA-95CA-C5AB0DC85B11.

Sur la base de l' exemple Sec-WebSocket-Key fourni ci-dessus, voici le Sec-WebSocket-Accepter l'en-tête renvoyé par le serveur :

```
Sec-WebSocket-Accept : s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

> Si l' en-tête Sec-WebSocket-Key est manquant dans la négociation initiée par le client, le serveur arrête de traiter la requête et renvoie une réponse HTTP avec un code d'erreur approprié (400 Bad Request, par exemple). S'il y a un problème avec la valeur de Sec-WebSocket-Accept, ou si l'en-tête est manquant dans la réponse du serveur, la connexion WebSocket ne sera pas établie (le client échoue à établir la connexion).

Cadres de messages

Après une ouverture réussie, le client et le serveur peuvent utiliser la connexion WebSocket pour échanger des messages en mode duplex intégral. Un message WebSocket se compose d'une ou plusieurs trames (consultez la section Bit FIN et fragmentation plus loin dans ce chapitre pour plus de détails sur les messages multitrames).

La trame WebSocket a une syntaxe binaire et contient plusieurs informations, comme le montre la figure suivante :

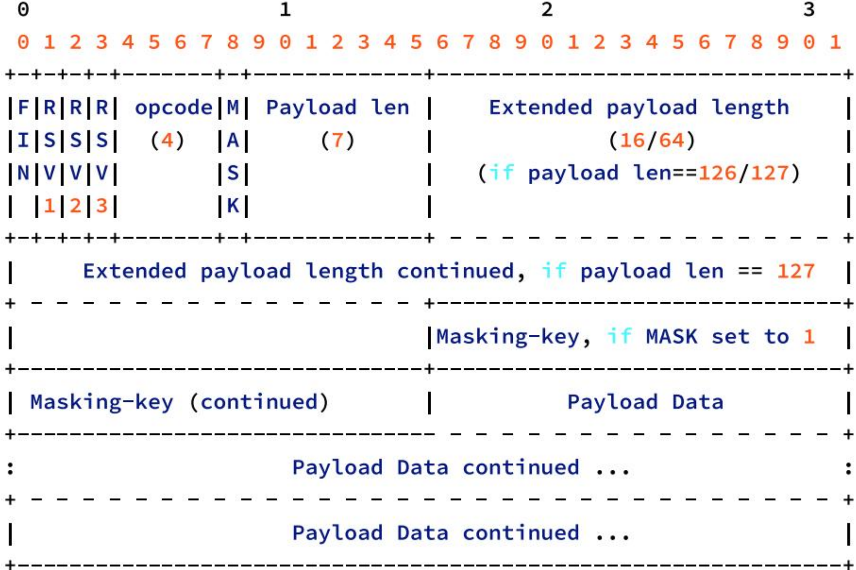

Figure 2.3 : Anatomie d'une trame WebSocket

Résumons-les rapidement :

- Bit FIN - indique si la trame est le fragment final d'un message WebSocket.
- RSV 1, 2, 3 - réservé aux extensions WebSocket.
- Opcode - détermine comment interpréter les données de charge utile.
- Masque - indique si la charge utile est masquée ou non.
- Clé de masquage - clé utilisée pour démasquer les données de charge utile.
- Longueur de la charge utile (étendue) - la longueur de la charge utile.
- Données de charge utile : se composent de données d’application et d’extension.

Nous allons maintenant examiner plus en détail tous ces éléments constitutifs d’une trame WebSocket.

Bit FIN et fragmentation

Il existe de nombreux scénarios dans lesquels la fragmentation d'un message WebSocket en plusieurs trames est nécessaire (ou du moins souhaitable). Par exemple, la fragmentation est souvent utilisée pour améliorer les performances. Sans fragmentation, un point de terminaison devrait mettre en mémoire tampon l'intégralité du message avant de l'envoyer. Avec la fragmentation, le point de terminaison peut choisir une mémoire tampon de taille raisonnable et, lorsqu'elle est pleine, envoyer les trames suivantes en guise de continuation. Le point de terminaison de réception assemble ensuite les trames pour recréer le message WebSocket.

Selon la RFC 645519, un autre cas d'utilisation de la fragmentation est représenté par le multiplexage, où « [...] il n'est pas souhaitable qu'un message volumineux sur un canal logique monopolise le canal de sortie, le multiplexage doit donc être libre de diviser le message en fragments plus petits pour mieux partager le canal de sortie. »

> Toutes les trames de données qui composent un message WebSocket doivent être du même type (texte ou binaire) ; vous ne pouvez pas avoir un message fragmenté composé à la fois de trames texte et binaires. Cependant, un message WebSocket fragmenté peut inclure des trames de contrôle. Consultez la section Opcodes plus loin dans ce chapitre pour plus de détails sur les types de trames.

Voyons maintenant quelques exemples rapides pour illustrer la fragmentation. Voici à quoi pourrait ressembler un message à trame unique :

```
0x81 0x05 0x48 0x65 0x6c 0x6c 0x6f (contient « Bonjour »)
```

En comparaison, avec la fragmentation, le même message ressemblerait à ceci :

```
0x01 0x03 0x48 0x65 0x6c (contient « Hel »)
0x80 0x02 0x6c 0x6f (contient « lo »)
```

Le protocole WebSocket permet la fragmentation via le premier bit de la trame WebSocket, le bit FIN , qui indique si la trame est le fragment final d'un message. Si c'est le cas, le bit FIN doit être défini sur 1. Toute autre trame doit avoir le bit FIN vide.

19 RFC 6455 : Le protocole WebSocket

VRS 1-3

RSV1, RSV2 et RSV3 sont des bits réservés. Ils doivent être 0, sauf si une extension a été négociée pendant la négociation d'ouverture qui définit des valeurs différentes de zéro. Consultez la section Extensions de ce chapitre pour plus de détails.

Codes d'opération

Chaque trame possède un opcode qui détermine comment interpréter les données de charge utile de cette trame. Les opcodes standard actuellement utilisés sont définis par la RFC 6455 et maintenus par l'IANA20.

CODE OPÉRATIONNEL	DESCRIPTION
0	Cadre de continuation ; continue la charge utile de l'image précédente.
1	Indique un cadre de texte (données texte UTF-8).
2	Indique une trame binaire.
3-7	Réservé aux trames de données personnalisées.
8	Cadre de fermeture de connexion ; conduit à la fin de la connexion.
9	Une trame ping. Sert de mécanisme de battement de cœur pour garantir que la connexion est toujours active. Le récepteur doit répondre avec une trame pong.
10	Une trame pong. Sert de mécanisme de battement de cœur pour garantir que la connexion est toujours active. Envoyée en réponse après avoir reçu une trame ping.
11-15	Réservé aux cadres de contrôle personnalisés.

Tableau 2.2 : codes d'opération de trame

> 8 (Close), 9 (Ping) et 10 (Pong) sont connus sous le nom de trames de contrôle et sont utilisés pour communiquer l'état de la connexion WebSocket.

20 Registre des codes d'opération IANA WebSocket

Masquage

Chaque trame WebSocket envoyée par le client au serveur doit être masquée à l'aide d'une clé de masquage aléatoire (valeur 32 bits). Cette clé est contenue dans la trame et est utilisée pour masquer les données de charge utile. Cependant, lorsque les données circulent dans l'autre sens, le serveur ne doit masquer aucune trame qu'il envoie au client.

> Un bit de masquage défini sur 1 indique que la trame concernée est masquée (et contient donc une clé de masquage). Le serveur fermera la connexion WebSocket s'il reçoit une trame non masquée.

Côté serveur, les trames reçues du client doivent être démasquées avant tout traitement ultérieur. Voici un exemple de la manière dont vous pouvez procéder :

```
var unmask = fonction(masque, tampon) {
      var payload = nouveau Buffer(buffer.length);
      pour (var i=0; i<buffer.length; i++) {
            charge utile[i] = masque[i % 4] ^ tampon[i];
      }
      retour de la charge utile ;
}
```

> Le masquage est utilisé comme mécanisme de sécurité qui aide à prévenir l'empoisonnement du cache.

Longueur de la charge utile

Le protocole WebSocket encode la longueur des données de charge utile à l'aide d'un nombre variable d'octets :

- Pour les charges utiles < 126 octets, la longueur est compressée dans les deux premiers octets d'en-tête de trame.
- Pour les charges utiles de 126 octets, deux octets d'en-tête supplémentaires sont utilisés pour indiquer la longueur.
- Si la charge utile est de 127 octets, huit octets d'en-tête supplémentaires sont utilisés pour indiquer sa longueur.

Le protocole WebSocket prend en charge deux types de données utiles : texte (texte Unicode UTF-8) et binaire. En JavaScript, les données texte sont appelées String, tandis que les données binaires sont représentées par les classes ArrayBuffer et Blob . Pour plus de détails sur l'envoi et la réception de données via WebSocket, ainsi que des exemples d'utilisation, consultez le chapitre 3 : L'API WebSocket.

> Les données de charge utile se composent de données d'application et potentiellement de données d'extension (à condition que les extensions aient été négociées lors de la négociation d'ouverture).

Le type de charge utile de chaque trame est indiqué via un opcode 4 bits (1 pour le texte ou 2 pour le binaire).

Poignée de main de clôture

Comparé à la poignée de main d'ouverture, la poignée de main de fermeture est un processus beaucoup plus simple. Vous l'initialisez en envoyant une trame de fermeture avec un opcode de 8. En plus de l'opcode, la trame de fermeture peut contenir un corps qui indique la raison de la fermeture. Ce corps se compose d'un code d'état (entier) et d'une chaîne codée en UTF-8 (la raison).

Les codes d'état standard qui peuvent être utilisés lors de la négociation de clôture sont définis par la RFC 6455 ; des codes de clôture personnalisés supplémentaires peuvent être enregistrés auprès de l'IANA[21].

CODE D'ÉTAT	NOM	DESCRIPTION
0-999	N / A	Les codes inférieurs à 1000 ne sont pas valides et ne peuvent pas être utilisés.
1000	Fermeture normale	Indique une fermeture normale, ce qui signifie que le but pour lequel la connexion WebSocket a été établie a été accompli.
1001	Partir	Doit être utilisé lors de la fermeture de la connexion et lorsqu'il n'y a aucune attente qu'une connexion de suivi soit tentée (par exemple, l'arrêt du serveur ou la navigation du navigateur hors de la page).
1002	Erreur de protocole	Le point de terminaison met fin à la connexion en raison d'une erreur de protocole.

21 Registre des numéros de code de fermeture IANA WebSocket

CODE D'ÉTAT	NOM	DESCRIPTION
1003	Données non prises en charge	La connexion est interrompue car le point de terminaison a reçu des données d'un type qu'il ne peut pas gérer (par exemple, un point de terminaison texte uniquement recevant des données binaires).
1004	Réservé	Réservé. Une signification pourrait être définie dans le futur.
1005	Aucun statut reçu	Utilisé par les applications et l'API WebSocket pour indiquer qu'aucun code d'état n'a été reçu, bien qu'un code soit attendu.
1006	Fermeture anormale	Utilisé par les applications et l'API WebSocket pour indiquer qu'une connexion a été fermée de manière anormale (par exemple, sans envoyer ni recevoir de trame de fermeture).
1007	Charge utile invalide données	Le point de terminaison met fin à la connexion car il a reçu un message contenant des données incohérentes (par exemple, des données non UTF-8 dans un message texte).
1008	Violation de la politique	Le point de terminaison met fin à la connexion car il a reçu un message qui viole sa politique. Il s'agit d'un code d'état générique. Il doit être utilisé lorsque les autres codes d'état ne conviennent pas ou s'il est nécessaire de masquer des détails spécifiques concernant la politique.
1009	Message trop gros	Le point de terminaison met fin à la connexion en raison de la réception d'une trame de données trop volumineuse pour être traitée.
1010	Obligatoire extension	Le client met fin à la connexion car le serveur n'a pas réussi à négocier une extension lors de la négociation d'ouverture.
1011	Erreur interne	Le serveur met fin à la connexion car il a rencontré une condition inattendue qui l'a empêché de répondre à la demande.
1012	Redémarrage du service	Le serveur met fin à la connexion car il redémarre.
1013	Réessayez plus tard	Le serveur met fin à la connexion en raison d'une condition temporaire, par exemple une surcharge.
1014	Mauvaise passerelle	Le serveur agissait comme une passerelle ou un proxy et a reçu une réponse non valide du serveur en amont. Similaire au code d'état HTTP 502 Bad Gateway.
1015	Poignée de main TLS	Réservé. Indique que la connexion a été fermée en raison à un échec d'exécution d'une négociation TLS (par exemple, le serveur le certificat ne peut pas être vérifié).

CODE D'ÉTAT	NOM	DESCRIPTION
1016-1999	N / A	Réservé à une utilisation future par la norme WebSocket.
2000-2999	N / A	Réservé pour une utilisation future par les extensions WebSocket.
3000-3999	N / A	Réservé à l'usage des bibliothèques, des frameworks et des applications. Possibilité d'inscription auprès de l'IANA selon le principe du premier arrivé, premier servi.
4000-4999	N / A	Gamme réservée à un usage privé dans les applications.

Tableau 2.3 : Codes d'état de clôture de la négociation

Le client et le serveur peuvent tous deux initier la négociation de fermeture. À la réception d'une trame de fermeture, un point de terminaison (client ou serveur) doit envoyer une trame de fermeture en guise de réponse (faisant écho au code d'état reçu).

> Une fois qu'une trame fermée a été envoyée, aucune autre trame de données ne peut passer via la connexion WebSocket.

Une fois qu'un point de terminaison a envoyé et reçu une trame de fermeture , la négociation de fermeture est terminée et la connexion WebSocket est considérée comme fermée.

Sous-protocoles

Les sous-protocoles (terme utilisé dans la RFC 6455) sont des protocoles de niveau application superposés au protocole WebSocket brut. Ils permettent de définir des formats spécifiques et une sémantique de niveau supérieur pour les échanges de données entre le client et le serveur. Les sous-protocoles peuvent garantir un accord non seulement sur la manière dont les données sont structurées, mais également sur la manière dont la communication doit commencer, se poursuivre et finalement se terminer.

Les sous-protocoles peuvent être regroupés en trois catégories principales :

- Protocoles enregistrés. Il s'agit des protocoles enregistrés auprès de l'IANA22.
- Protocoles ouverts. Protocoles ouverts, tels que Message Queuing Telemetry Transport (MQTT)23 ou protocole de message orienté texte simple (STOMP)24.
- Protocoles personnalisés. Se réfère à des bibliothèques open source ou à des solutions propriétaires introduisant leur saveur spécifique de communications basées sur WebSocket.

Les sous-protocoles sont négociés lors de la négociation d'ouverture. Le client utilise l' en-tête Sec-WebSocket-Protocol pour transmettre un ou plusieurs sous-protocoles séparés par des virgules, comme illustré dans cet exemple :

```
Protocole Sec-WebSocket : amqp, v12.stomp
```

À condition qu'il comprenne les sous-protocoles transmis dans la requête client, le serveur doit en choisir un (et un seul) et le renvoyer avec l' en-tête Sec-WebSocket-Protocol . À partir de ce moment, le client et le serveur peuvent communiquer via le sous-protocole négocié.

> Si le serveur n'est d'accord avec aucun des sous-protocoles suggérés, l' en-tête Sec-WebSocket-Protocol ne sera pas inclus dans la réponse.

22 Registre des noms de sous-protocoles WebSocket de l'IANA

23 Kayla Matthews, MQTT : une plongée conceptuelle en profondeur

24 Le protocole de messagerie orienté texte simple (STOMP)

Extensions

Les extensions sont nommées ainsi car elles étendent le protocole WebSocket. Elles peuvent être utilisées pour ajouter de nouveaux opcodes, champs de données et fonctionnalités supplémentaires (multiplexage, par exemple) au protocole WebSocket standard.

Au moment de la rédaction de cet article, seules quelques extensions sont enregistrées auprès de l'IANA25, comme permessage-deflate, qui compresse la partie des données de charge utile des trames WebSocket. Si vous souhaitez développer votre propre extension, vous pouvez utiliser un framework open source comme websocket-extensions26.

Les extensions sont négociées lors de la négociation d'ouverture. Le client utilise l' en-tête Sec-Websocket-Extensions pour transmettre les extensions qu'il souhaite utiliser, comme illustré dans cet exemple :

```
Extensions Sec-WebSocket : permessage-deflate, mon-extension-personnalisée
```

À condition qu'il prenne en charge les extensions envoyées dans la requête client, le serveur doit les inclure dans la réponse, avec l' en-tête Sec-WebSocket-Extensions . À partir de ce moment, le client et le serveur peuvent communiquer via WebSockets en utilisant les extensions qu'ils ont négociées.

Sécurité

Dans cette section, nous aborderons certains des mécanismes que vous pouvez utiliser pour sécuriser les connexions WebSocket et les communications effectuées via le protocole WebSocket. Cette section n'est en aucun cas exhaustive ; elle vise uniquement à fournir un aperçu de haut niveau de plusieurs considérations liées à la sécurité. Le sujet complexe de la sécurité des WebSockets sera traité plus en détail dans les prochaines versions de ce livre.

Commençons par l' en-tête Origin , qui est envoyé par tous les clients de navigateur (facultatif pour les non-navigateurs) au serveur lors de la négociation d'ouverture. L' en-tête Origin est essentiel pour sécuriser la communication entre domaines. Plus précisément, si l' origine indiquée est inacceptable, le serveur peut faire échouer la négociation (généralement en renvoyant un code d'état HTTP 403 Forbidden). Cette capacité peut être extrêmement utile pour atténuer les attaques par déni de service (DoS).

25 Registre des noms d'extensions WebSocket IANA

26 Le framework d'extensions WebSocket

En parlant des en-têtes utilisés lors de la négociation d'ouverture, nous devons également mentionner Sec-WebSocket-Key et Sec-WebSocket-Accept. En un mot, le but de ces en-têtes est de protéger les serveurs WebSocket sans méfiance des attaques inter-protocoles initiées par des clients non-WebSocket. Ensemble, Sec-WebSocket-Key et Sec-WebSocket-Accept garantissent que le client et le serveur peuvent, en fait, communiquer via WebSockets. En cas de problème impliquant ces deux en-têtes (par exemple, Sec-WebSocket-Key est absent de la demande du client), la connexion WebSocket ne sera pas établie.

> Le protocole WebSocket ne prescrit aucune méthode particulière permettant aux serveurs d'authentifier les clients. Par exemple, vous pouvez gérer l'authentification lors de la négociation d'ouverture, en utilisant des en-têtes de cookie. Une autre option consiste à gérer l'authentification (et l'autorisation) au niveau de l'application, en utilisant des techniques telles que les jetons Web JSON27.

Jusqu'à présent, nous avons couvert les mécanismes de sécurité utilisés lors de l'établissement de la connexion. Voyons maintenant quelques aspects qui ont un impact sur la sécurité lors de l'échange de données entre le client et le serveur. Tout d'abord, pour réduire le risque d'attaques de type "man-in-the-middle" et d'écoute clandestine (en particulier lors de l'échange de données critiques et sensibles), il est recommandé d'utiliser le schéma URI wss , qui utilise TLS pour crypter la connexion, tout comme https.

Nous avons déjà parlé des trames de messages dans ce chapitre et mentionné que les trames envoyées par le client au serveur doivent être masquées à l'aide d'une clé de masquage aléatoire (valeur 32 bits). Cette clé est contenue dans la trame et est utilisée pour masquer les données de charge utile. Les trames doivent être démasquées par le serveur avant tout traitement ultérieur.
Le masquage permet de différencier le trafic WebSocket du trafic HTTP, ce qui est particulièrement utile lorsque des serveurs proxy sont impliqués. En effet, certains serveurs proxy peuvent ne pas « comprendre » le protocole WebSocket et, sans le masque, ils pourraient le confondre avec le trafic HTTP normal ; cela pourrait entraîner toutes sortes de problèmes, comme l'empoisonnement du cache.

27 RFC 7519 : jeton Web JSON (JWT)

L'API WebSocket

Ce chapitre vous présente l'interface de programmation d'application (API) WebSocket, qui étend le protocole WebSocket aux applications Web. L'API WebSocket permet une communication pilotée par événements via une connexion persistante. Cela vous permet de créer des applications Web véritablement en temps réel et moins gourmandes en ressources sur le client et le serveur par rapport aux techniques HTTP.

Dans les sections suivantes, nous examinerons les composants constitutifs de l'API WebSocket : ses événements, ses méthodes et ses propriétés.

Aperçu

Définie dans la norme HTML Living Standard[28], l'API WebSocket est une technologie qui permet d'ouvrir un canal de communication bidirectionnel et duplex intégral persistant entre un client Web et un serveur Web. L'interface WebSocket vous permet d'envoyer des messages de manière asynchrone à un serveur et de recevoir des réponses basées sur des événements sans avoir à interroger les mises à jour.

Presque tous les navigateurs modernes prennent en charge l' API WebSocket[29]. De plus, il existe de nombreux frameworks et bibliothèques, à la fois open source et commerciaux, qui implémentent les API WebSocket. Consultez la section Ressources de ce livre pour plus de détails.

Dans le reste de ce chapitre, nous aborderons les fonctionnalités principales de l'API WebSocket. Comme vous le verrez, elle est intuitive, conçue dans un souci de simplicité et facile à utiliser.

28 sockets Web, norme HTML vivante
29 Puis-je utiliser les WebSockets ?

Le serveur WebSocket

Un serveur WebSocket peut être écrit dans n'importe quel langage de programmation côté serveur compatible avec les sockets Berkeley30. Le serveur écoute les connexions WebSocket entrantes à l'aide d'un socket TCP standard. Une fois la négociation d'ouverture terminée, le serveur doit être capable d'envoyer, de recevoir et de traiter les messages WebSocket.

Consultez le chapitre 4 : Création d'une application Web avec WebSockets pour savoir comment créer votre propre serveur WebSocket dans Node.js.

Le constructeur WebSocket

Pour démarrer avec l'API WebSocket côté client, la première chose à faire est d'instancier un objet WebSocket , qui tentera automatiquement d'ouvrir la connexion au serveur :

```
const socket = new WebSocket('wss://exemple.org');
```

Le constructeur WebSocket contient un paramètre obligatoire : l' URL du serveur WebSocket. De plus, le paramètre facultatif protocols peut également être inclus, pour indiquer un ou plusieurs sous-protocoles WebSocket (protocoles au niveau de l'application) qui peuvent être utilisés pendant la communication client-serveur :

```
const socket = new WebSocket('wss://example.org', 'myCustomProtocol');
```

Une fois l' objet WebSocket créé et la connexion établie, le client peut commencer à échanger des données avec le serveur.

> L'instanciation de l' objet WebSocket initie essentiellement la négociation d'ouverture
> nous l'avons mentionné dans le chapitre précédent.

30 Prises Berkeley

Evénements

La programmation WebSocket suit un modèle de programmation asynchrone et piloté par les événements. Tant qu'une connexion WebSocket est ouverte, le client et le serveur écoutent simplement les événements afin de gérer les données entrantes et les changements d'état de connexion (sans avoir besoin d'interrogation).

L'API WebSocket prend en charge quatre types d'événements :

- ouvrir
- message
- erreur
- fermer

> En JavaScript, les événements WebSocket peuvent être gérés à l'aide des propriétés « onevent » (par exemple, onopen est utilisé pour gérer les événements ouverts). Vous pouvez également utiliser la méthode addEventListener() . Dans tous les cas, votre code fournira des rappels qui s'exécuteront à chaque fois qu'un événement est déclenché.

Ouvrir

L' événement open est déclenché lorsqu'une connexion WebSocket est établie. Il indique que la négociation d'ouverture entre le client et le serveur a réussi et que la connexion WebSocket peut désormais être utilisée pour envoyer et recevoir des données. Voici un exemple d'utilisation :

```
// Créer une connexion WebSocket
const socket = new WebSocket('wss://exemple.org');

// Connexion ouverte
socket.onopen = fonction(e) {
    console.log('Connexion ouverte !');
};
```

Message

L' événement de message est déclenché lorsque des données sont reçues via un WebSocket. Les messages peuvent contenir des chaînes (texte brut) ou des données binaires, et c'est à vous de décider comment ces données seront traitées et visualisées.

Voici un exemple de gestion d'un événement de message :

```
socket.onmessage = fonction(msg) {
    si(msg.data instanceof ArrayBuffer) {
        processArrayBuffer(msg.données);
    } autre {
        processText(msg.données);
    }
  }
```

Erreur

L' événement d'erreur est déclenché en réponse à des échecs ou problèmes inattendus (par exemple, certaines données n'ont pas pu être envoyées). Voici comment écouter les événements d'erreur :

```
socket.onerror = fonction(e) {
    console.log(' Échec WebSocket', e);
    gérerErreurs(e);
};
```

> Les erreurs entraînent la fermeture de la connexion WebSocket, donc un événement d'erreur est toujours suivi de peu par un événement de fermeture.

Fermer

L' événement de fermeture se déclenche lorsque la connexion WebSocket se ferme. Cela peut être dû à diverses raisons, telles qu'un échec de connexion, une négociation de fermeture réussie ou des erreurs TCP. Une fois la connexion terminée, le client et le serveur ne peuvent plus envoyer ni recevoir messages.

Voici comment écouter un événement proche :

```
socket.onclose = fonction(e) {
    console.log('Connexion fermée', e);
};
```

> Vous pouvez déclencher manuellement l'appel de l' événement de fermeture en exécutant la fonction close() méthode.

Méthodes

L'API WebSocket prend en charge deux méthodes : send() et close().

envoyer()

Une fois la connexion établie, vous êtes presque prêt à commencer à envoyer et à recevoir des messages vers et depuis le serveur WebSocket. Mais avant de faire cela, vous devez d'abord vous assurer que la connexion est ouverte et prête à recevoir des messages. Vous pouvez y parvenir de deux manières principales.

La première option consiste à déclencher la méthode send() à partir du gestionnaire d'événements onopen , comme illustré dans l'exemple suivant :

```
socket.onopen = fonction(e) {
    socket.send(JSON.stringify({'msg': 'charge utile'}));
}
```

La deuxième méthode consiste à vérifier la propriété readyState et à choisir d'envoyer des données uniquement lorsque la connexion WebSocket est ouverte :

```
fonction processEvent(e) {
        si(socket.readyState === WebSocket.OPEN) {
                // Socket ouvert, envoyer !
                socket.envoyer(e);
        } autre {
                // Afficher une erreur, la mettre en file d'attente pour un envoi ultérieur, etc.
        }
}
```

Les deux extraits de code ci-dessus montrent comment envoyer des messages texte (chaîne). Cependant, en plus des chaînes, vous pouvez également envoyer des données binaires (Blob ou ArrayBuffer), comme illustré dans cet exemple :

```
var tampon = nouveau ArrayBuffer(128);
socket.send(tampon);

var intview = new Uint32Array(tampon);
socket.envoyer(intview);

var blob = nouveau Blob([tampon]);
socket.envoyer(blob);
```

Après avoir envoyé un ou plusieurs messages, vous pouvez laisser la connexion WebSocket ouverte pour d'autres échanges de données, ou appeler la méthode close() pour y mettre fin.

fermer()

La méthode close() est utilisée pour fermer la connexion WebSocket (ou la tentative de connexion).
Il s'agit essentiellement de l'équivalent de la poignée de main de fermeture que nous avons abordée précédemment, au chapitre 2. Une fois cette méthode appelée, aucune autre donnée ne peut être envoyée ou reçue via le WebSocket. connexion.

> Si la connexion est déjà fermée, l'appel de la méthode close() ne fait rien.

Voici l'exemple le plus basique d'appel de la méthode close() :

```
socket.close();
```

En option, vous pouvez passer deux arguments avec la méthode close() :

- code. Une valeur numérique indiquant le code d'état expliquant pourquoi la connexion est étant fermé. Voir la section Poignée de main de clôture au chapitre 2 pour plus de détails sur tous les codes d'état qui peuvent être utilisés.

- raison. Une chaîne lisible par l'homme expliquant pourquoi la connexion se ferme.

Voici un exemple d'appel de la méthode close() avec les deux paramètres facultatifs :

```
socket.close(1003, 'Type de données non pris en charge !');
```

Propriétés

L' objet WebSocket expose plusieurs propriétés contenant des détails sur la connexion WebSocket.

Type binaire

La propriété binaryType contrôle le type de données binaires reçues via la connexion WebSocket. La valeur par défaut est blob ; en outre, les WebSockets prennent également en charge tampon de tableau.

montant mis en mémoire tampon

Propriété en lecture seule qui renvoie le nombre d'octets de données en file d'attente pour transmission mais pas encore envoyés. La valeur de bufferedAmount est réinitialisée à zéro une fois que toutes les données en file d'attente ont été envoyé.

bufferedAmount est particulièrement utile lorsque l'application cliente transporte de grandes quantités de données vers le serveur. Même si l'appel de send() est instantané, la transmission de ces données sur Internet ne l'est pas. Les navigateurs mettront en mémoire tampon les données sortantes pour le compte de votre application cliente. La propriété bufferedAmount est utile pour garantir que toutes les données sont envoyées avant de fermer une connexion ou d'effectuer votre propre limitation côté client.

Vous trouverez ci-dessous un exemple d'utilisation de bufferedAmount pour envoyer des mises à jour toutes les secondes et ajuster en conséquence si le réseau ne peut pas gérer le débit :

```
// Taille maximale du tampon : 10 k.
var SEUIL = 10240;

// Créer une nouvelle connexion WebSocket
const socket = new WebSocket('wss://exemple.org');

// Écoutez l'événement d'ouverture
socket.onopen = fonction () {
    // Tenter d'envoyer une mise à jour toutes les secondes.
    setInterval(fonction () {
        // Envoyer uniquement si le tampon n'est pas plein
        si (socket.bufferedAmount < SEUIL) {
            socket.send(getApplicationState());
        }
    }, 1000);
};
```

extensions

Propriété en lecture seule qui renvoie le nom des extensions WebSocket qui ont été négociées entre le client et le serveur lors de la négociation d'ouverture.

> Si aucune extension n'a été négociée lors de l'établissement de la connexion, la propriété extensions renvoie une chaîne vide.

Propriétés de « onevent »

Ces propriétés sont appelées pour exécuter le code du gestionnaire associé chaque fois qu'un événement WebSocket est déclenché. Il existe quatre types de propriétés « onevent », une pour chaque type d'événement :

PROPRIÉTÉ	DESCRIPTION
ouvert	Appelé lorsque la propriété readyState de la connexion WebSocket passe à 1 ; cela indique que la connexion est ouverte et prête à envoyer et à recevoir données.
surmessage	Appelé lorsqu'un message est reçu du serveur.
une erreur	Est appelé lorsqu'un événement d'erreur se produit, affectant la connexion WebSocket.
fermer	Appelé avec un événement de fermeture lorsque la connexion WebSocket est readyState la propriété passe à 3 ; cela indique que la connexion est fermée.

Tableau 3.1 : propriétés de « onevent »

protocole

Propriété en lecture seule renvoyant le nom du sous-protocole WebSocket qui a été négocié pour la communication entre le client et le serveur lors de l'ouverture poignée de main.

> Les sous-protocoles sont spécifiés via le paramètre protocoles lors de la création de l'objet WebSocket (voir la section Constructeur WebSocket plus haut dans ce chapitre pour plus de détails). Si aucun protocole n'est spécifié lors de l'établissement de la connexion, la propriété protocole renverra une chaîne vide.

État prêt

Propriété en lecture seule qui renvoie l'état actuel de la connexion WebSocket. Le tableau ci-dessous indique les valeurs que vous pouvez voir reflétées par cette propriété, ainsi que leur signification :

VALEUR	ÉTAT	DESCRIPTION
0	DE LIAISON	Le socket a été créé, mais la connexion n'est pas encore établie ouvrir.
1	OUVRIR	La connexion est ouverte et prête à communiquer.
2	CLÔTURE	La connexion est en cours de fermeture.
3	FERMÉ	La connexion est fermée ou n'a pas pu être ouverte.

Tableau 3.2 : États de connexion WebSocket

> La valeur de readyState changera au fil du temps. Il est recommandé de la vérifier périodiquement pour comprendre la durée de vie et le cycle de vie du WebSocket connexion.

URL

Propriété en lecture seule qui renvoie l'URL absolue du WebSocket, telle que résolue par le constructeur.

CHAPITRE 4

Créer une application Web avec WebSockets

La version initiale de ce chapitre a été écrite et publiée par Jo Franchetti[31]

Dans ce chapitre, nous verrons comment créer une application Web en temps réel avec WebSockets et Node.js : une démonstration interactive de partage de position de curseur. C'est le genre de projet qui nécessite une communication bidirectionnelle et instantanée entre le client et le serveur, le type de cas d'utilisation où la technologie WebSocket brille vraiment.

Clients et serveurs WebSocket

Pour exploiter la technologie WebSocket côté serveur, une application backend est nécessaire. Pour notre démonstration, nous utiliserons Node.JS, un environnement d'exécution JavaScript asynchrone, léger et efficace, piloté par événements. Node.js est un excellent choix pour créer des applications Web évolutives en temps réel et gérer plusieurs centaines de connexions WebSocket simultanées.
Nous verrons comment implémenter deux bibliothèques Node.js différentes en tant que serveur WebSocket : ws et SockJS.

L'utilisation de WebSockets en frontend est assez simple, via l'API WebSocket intégrée à tous les navigateurs modernes (nous utiliserons cette API côté client dans la première partie de la démonstration, aux côtés de ws côté serveur). De plus, il existe de nombreuses bibliothèques et solutions implémentant la technologie WebSocket à la fois côté client et côté serveur. Cela inclut SockJS, que nous aborderons dans la deuxième partie de la démonstration.

Pour plus de détails sur les implémentations client et serveur WebSocket, consultez les ressources section.

31 Jo Franchetti, WebSockets et Node.js — tester WS et SockJS en créant une application Web

ws — une bibliothèque WebSocket Node.js

ws32 est un serveur WebSocket pour Node.js. Il est de niveau assez bas : vous écoutez les demandes de connexion entrantes et répondez aux messages bruts sous forme de chaînes ou de tampons d'octets.

Afin de démontrer comment configurer des WebSockets avec Node.js et ws, nous allons créer une application de démonstration qui partage les positions du curseur des utilisateurs en temps réel. Nous vous expliquerons comment le créer ci-dessous.

Créer une démonstration interactive de partage de position de curseur avec ws

Il s'agit d'une démonstration permettant de créer une icône de curseur colorée pour chaque utilisateur connecté. Lorsqu'ils déplacent leur souris, leur icône de curseur se déplace sur l'écran et s'affiche également en mouvement sur l'écran de chaque utilisateur connecté. Cela se produit en temps réel, au fur et à mesure que la souris est déplacée.

Configuration du serveur WebSocket

Tout d'abord, nécessitez la bibliothèque ws et utilisez la méthode WebSocket.Server pour créer un nouveau serveur WebSocket sur le port 7071 (ou tout autre port de votre choix) :

```
const WebSocket = require('ws');
const wss = nouveau WebSocket.Server({ port: 7071 });
```

> Par souci de concision, nous l'appelons wss dans notre code. Toute ressemblance avec les WebSockets sécurisés (souvent appelés wss) est une coïncidence.

Ensuite, créez une carte pour stocker les métadonnées d'un client (toutes les données que nous souhaitons associer à un Client WebSocket) :

```
const clients = nouvelle carte();
```

32 ws : une bibliothèque WebSocket Node.js

fonction sera déclenchée chaque fois qu'un nouveau client WebSocket se connecte au serveur :

```
wss.on('connexion', (ws) => {
        const id = uuidv4();
        const couleur = Math.floor(Math.random() * 360);
        const métadonnées = { id, couleur };

        clients.set(ws, métadonnées);
```

Chaque fois qu'un client se connecte, nous générons un nouvel identifiant unique, qui est utilisé pour l'identifier. Les clients se voient également attribuer une couleur de curseur à l'aide de Math.random(); cela génère un nombre compris entre 0 et 360, qui correspond à la valeur de teinte d'une couleur HSV. L'ID et la couleur du curseur sont ensuite ajoutés à un objet que nous appellerons métadonnées, et nous utilisons la carte pour les associer à notre instance ws WebSocket.

La carte est un dictionnaire — nous pouvons récupérer ces métadonnées en appelant get et en fournissant une instance WebSocket ultérieurement.

En utilisant l'instance WebSocket nouvellement connectée, nous nous abonnons au message de cette instance événement et fournir une fonction de rappel qui sera déclenchée chaque fois que ce client spécifique envoie un message au serveur.

```
ws.on('message', (messageAsString) => {
```

> Cet événement se produit sur l'instance WebSocket (ws) elle-même, et non sur le WebSocket. Instance de serveur (wss).

Chaque fois que notre serveur reçoit un message, nous utilisons JSON.parse pour obtenir le contenu du message et chargeons nos métadonnées client pour ce socket à partir de notre carte à l'aide de clients.get(ws).

Nous allons ajouter nos deux propriétés de métadonnées au message en tant qu'expéditeur et couleur :

```
message const = JSON.parse(messageAsString);
const métadonnées = clients.get(ws);

message.sender = métadonnées.id;
message.color = métadonnées.color;
```

Ensuite, nous transformons à nouveau notre message en chaîne et l'envoyons à chaque client connecté :

```
const sortant = JSON.stringify(message);

  [...clients.keys()].forEach((client) => {
     client.send(sortant); }); });
```

Enfin, lorsqu'un client ferme sa connexion, nous supprimons ses métadonnées de notre carte :

```
    ws.on("fermer", () =>
        { clients.delete(ws); });

});
```

En bas, nous avons une fonction pour générer un identifiant unique :

```
fonction uuidv4() {
   renvoie 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c)
{
       var r = Math.random() * 16 | 0, v = c == 'x' ? r : (r & 0x3 | 0x8); return v.toString(16); });



} console.log('wss up');
```

Cette implémentation de serveur effectue une multidiffusion, envoyant tout message reçu à tous les clients connectés.

Nous devons maintenant écrire du code côté client pour nous connecter au serveur WebSocket et transmettre la position du curseur de l'utilisateur lorsqu'il se déplace.

WebSockets côté client

Nous allons commencer avec quelques éléments HTML5 standard :

```
<!DOCTYPE html> <html
lang='fr'> <head> <meta

charset='UTF-8'> <meta http-equiv='Compatible
        X-UA' content='IE=edge'> <meta name='viewport' content='width=device-width, initial-
        scale=1.0'> <title>Document</title>
```

Ensuite, nous ajoutons une référence à une feuille de style et un fichier index.js que nous ajoutons en tant que module ES (en utilisant type="module").

```
        <link rel='stylesheet' href='style.css'>
        <script src='index.js' type='module'></script>
</tête>
```

Le corps contient un seul modèle HTML qui contient une image SVG d'un pointeur.

Nous allons utiliser JavaScript pour cloner ce modèle chaque fois qu'un nouvel utilisateur se connecte à notre serveur.

```
<body id='box'>
        <template id='curseur'>
                <svg viewBox='0 0 16.3 24.7' class='curseur'>
                        <path stroke='#000' stroke-linecap='rond' stroke -linejoin='rond' stroke-miterlimit='10'
d='M15.6
            15.6L.6.6v20.5l4.6-4.5 3.2 7.5
                        3.4-1.3-3-7.2z' />
                </svg>
        </modèle>
</body>
</html>
```

Ensuite, nous devons utiliser JavaScript pour nous connecter au serveur WebSocket :

```
(fonction asynchrone() {
        const ws = wait connectToServer();
        ...
```

Nous appelons la fonction connectToServer() , qui résout une promesse contenant le WebSocket connecté (la définition de la fonction sera écrite plus tard).

Une fois connecté, nous ajoutons un gestionnaire pour onmousemove au document.body. Le messageBody est très simple : il se compose des propriétés clientX et clientY actuelles de l'événement de mouvement de la souris (les coordonnées horizontales et verticales du curseur dans la fenêtre d'affichage de l'application).

Nous transformons cet objet en chaîne et l'envoyons via notre instance WebSocket ws désormais connectée en tant que texte du message :

```
document.body.onmousemove = (evt) => {
          messageBody = { x: evt.clientX, y: evt.clientY };
          ws.send(JSON.stringify(messageBody));
  };
```

Nous devons maintenant ajouter un autre gestionnaire, cette fois pour un événement onmessage à l'instance WebSocket ws. N'oubliez pas que chaque fois que le serveur WebSocket reçoit un message, il le transmet à tous les clients connectés.

Vous remarquerez peut-être que la syntaxe ici diffère légèrement du code WebSocket côté serveur. C'est parce que nous utilisons la classe WebSocket native du navigateur , plutôt que la bibliothèque ws .

```
ws.onmessage = (webSocketMessage) => {
        messageBody = JSON.parse(webSocketMessage.data);
        const cursor = getOrCreateCursorFor(messageBody);
        cursor.style.transform = `traduire(${messageBody.x}px, ${messageBody.y}px)`;

  };
```

Lorsque nous recevons un message via WebSocket, nous analysons la propriété data du message, qui contient les données stringifiées que le gestionnaire onmousemove a envoyées au serveur WebSocket, ainsi que les propriétés d'expéditeur et de couleur supplémentaires que le code côté serveur ajoute au message.

En utilisant le messageBody analysé, nous appelons getOrCreateCursorFor. Cette fonction renvoie un élément HTML qui fait partie du DOM. Nous verrons comment cela fonctionne plus tard.

Nous utilisons ensuite les valeurs x et y du messageBody pour ajuster la position du curseur à l'aide d'un Transformation CSS.

Notre code repose sur deux fonctions utilitaires. La première est connectToServer, qui ouvre une connexion à notre serveur WebSocket, puis renvoie une promesse qui se résout lorsque la propriété WebSocket readyState est égale à 1 - CONNECTED.

Cela signifie que nous pouvons simplement attendre cette fonction et nous saurons que nous avons une connexion WebSocket connectée et fonctionnelle.

```
fonction asynchrone connectToServer() {
        const ws = new WebSocket('ws://localhost:7071/ws');
        retourner une nouvelle promesse ((résoudre, rejeter) => {
                const timer = setInterval(() => {
                        si (ws.readyState === 1) {
                                clearInterval(minuterie)
                                résoudre(ws);
                        }
                }, 10);
        });
  }
```

Nous utilisons également notre fonction getOrCreateCursor. Cette fonction tente d'abord de trouver tout élément existant avec l'attribut de données HTML data-sender où la valeur est la même que la propriété sender dans notre message. Si elle en trouve un, nous savons que nous avons déjà créé un curseur pour cet utilisateur, et nous devons simplement le renvoyer pour que le code appelant puisse ajuster sa position.

```
fonction getOrCreateCursorFor(messageBody) {
        const sender = messageBody.sender;
        const existant = document.querySelector(`[data-sender='${sender}']`);
        si (existant) {
                retourner l'existant;
        }
```

Si nous ne trouvons pas d'élément existant, nous clonons notre modèle HTML, y ajoutons l'attribut data avec l' ID de l'expéditeur actuel et l'ajoutons au document.body avant de le renvoyer :

```
                const template = document.getElementById('curseur');
                const cursor = template.content.firstElementChild.cloneNode(true);
                const svgPath = cursor.getElementsByTagName('chemin')[0];

                cursor.setAttribute('expéditeur de données', expéditeur);
                svgPath.setAttribute('fill', `hsl(${messageBody.color}, 50%, 50%)`);
                document.body.appendChild(curseur);

                retourner le curseur;
        }

}) ();
```

Désormais, lorsque vous exécutez l'application Web, chaque utilisateur qui consulte la page aura un curseur qui apparaît sur les écrans de chacun, car nous envoyons les données à tous les clients à l'aide de WebSockets.

Exécution de la démo

Si vous avez suivi le didacticiel, vous pouvez exécuter :

```
> installation npm
> démarrer l'exécution de npm
```

Sinon, vous pouvez cloner une version fonctionnelle de la démo à partir de : https://github.com/ably-labs/ partage de curseur websockets.

```
> git clone https://github.com/ably-labs/WebSockets-cursor-sharing.git
> installation npm
> démarrer l'exécution de npm
```

Cette démo comprend deux applications : une application Web que nous servons via Snowpack33 et un serveur Web Node.js. La tâche de démarrage NPM lance à la fois l'API et le serveur Web.

La démo devrait ressembler à celle illustrée ci-dessous :

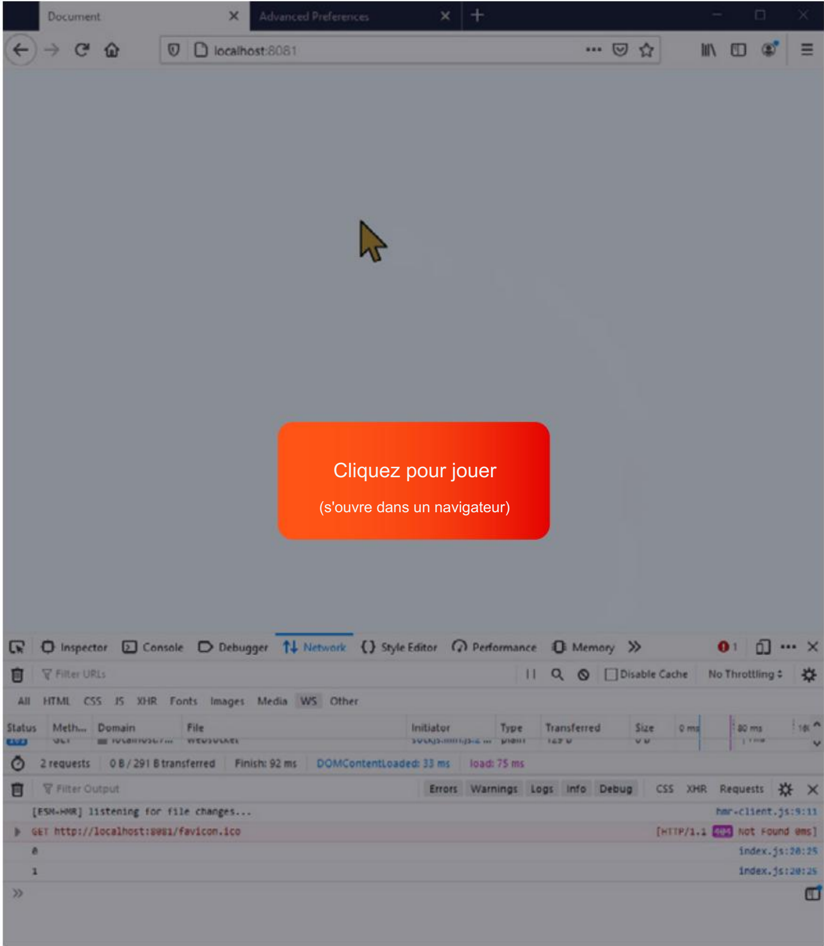

Figure 4.1 : Mouvement du curseur en temps réel grâce à la bibliothèque ws WebSockets

[33] Manteau neigeux

Cependant, si vous exécutez la démo dans un navigateur qui ne prend pas en charge les WebSockets (par exemple, IE9 ou une version antérieure), ou si vous êtes limité par des proxys d'entreprise particulièrement stricts, vous obtiendrez une erreur indiquant que le navigateur ne peut pas établir de connexion :

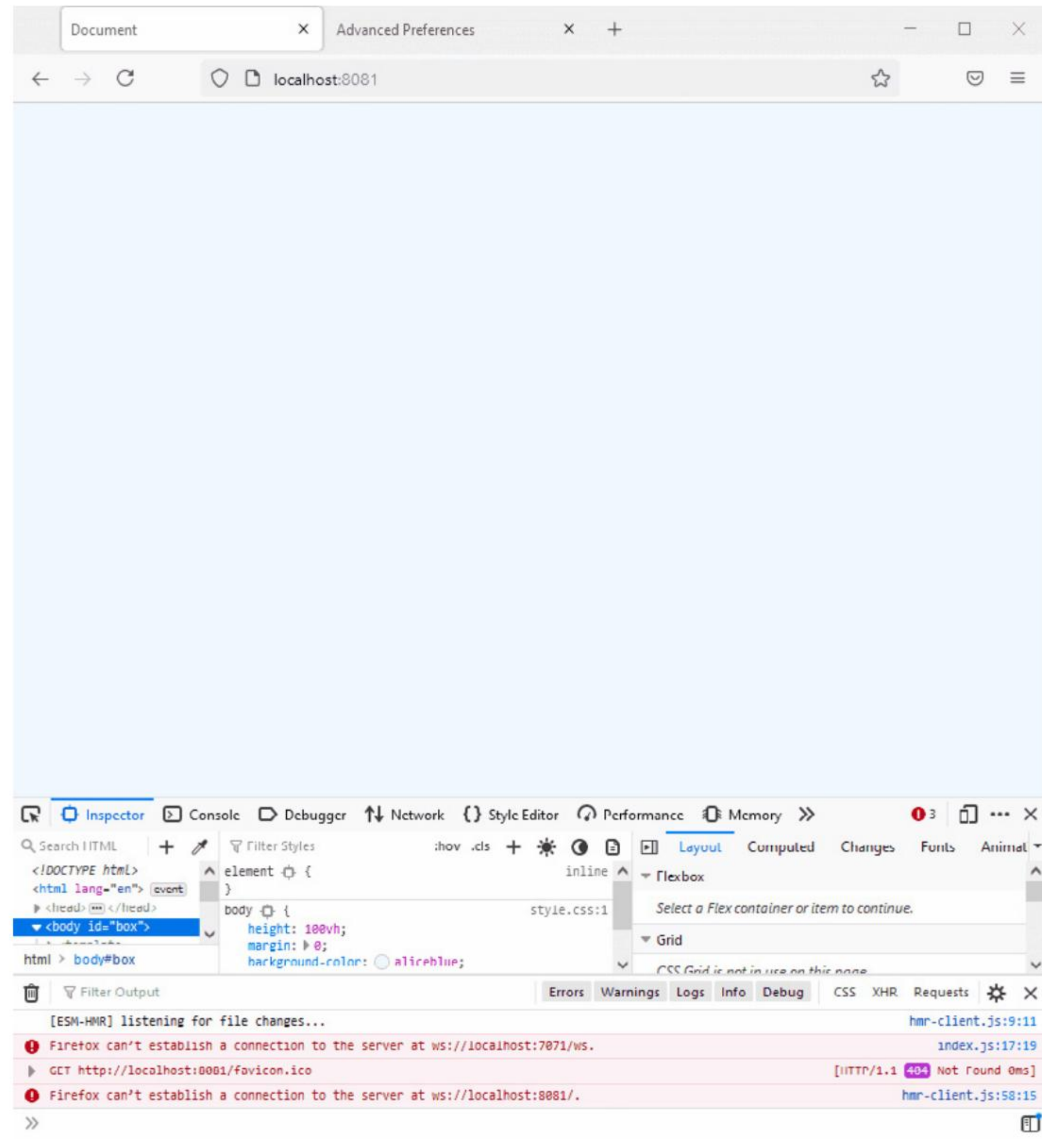

Figure 4.2 : Message d'erreur renvoyé par le navigateur lorsqu'une connexion WebSocket ne peut pas être établie

Cela est dû au fait que la bibliothèque ws n'offre aucun protocole de transfert de secours si les WebSockets ne sont pas disponibles. Si cela est une exigence pour votre projet, ou si vous souhaitez bénéficier d'un niveau de fiabilité de livraison plus élevé pour vos messages, vous aurez besoin d'une bibliothèque qui offre plusieurs protocoles de transfert, comme SockJS.

SockJS — une bibliothèque JavaScript pour fournir une communication de type WebSocket

SockJS est une bibliothèque qui imite l'API WebSocket native dans les navigateurs. De plus, elle reviendra à HTTP chaque fois qu'un WebSocket ne parvient pas à se connecter ou si le navigateur utilisé ne prend pas en charge les WebSockets. Comme ws, SockJS nécessite un équivalent serveur ; ses responsables fournissent à la fois une bibliothèque client JavaScript34 et une bibliothèque serveur Node.js35.

L'utilisation de SockJS dans le client est similaire à l'API WebSocket native, avec quelques petites différences. Nous pouvons remplacer ws dans la démo créée précédemment et utiliser SockJS à la place pour inclure la prise en charge de secours.

Mise à jour de la démonstration interactive de partage de position du curseur pour utiliser SockJS

Pour utiliser SockJS dans le client, nous devons d'abord charger la bibliothèque JavaScript SockJS à partir de leur CDN. Dans l'en-tête du document index.html que nous avons créé précédemment, ajoutez la ligne suivante au-dessus du script include de index.js :

```
<script src='https://cdn.jsdelivr.net/npm/sockjs-client@1/dist/sockjs.min.
js' defer></script>
```

Notez le mot-clé defer — il garantit que la bibliothèque SockJS est chargée avant index.js court.

Dans le fichier app/script.js , nous mettons ensuite à jour le JavaScript pour utiliser SockJS. Au lieu de l' objet WebSocket , nous allons maintenant utiliser un objet SockJS . Dans la fonction connectToServer , nous allons établir la connexion avec le serveur SockJS :

```
const ws = new SockJS('http://localhost:7071/ws');
```

SockJS nécessite un chemin de préfixe sur l'URL du serveur. Le reste de l' app/script.js

le fichier ne nécessite aucune modification.

34 Client SockJS

35 nœuds SockJS

Ensuite, nous devons mettre à jour le fichier API/script.js pour que notre serveur utilise SockJS. Cela signifie changer les noms de quelques hooks d'événements, mais l'API est très similaire à ws.

Tout d'abord, nous devons installer sockjs-node. Dans votre terminal, exécutez :

```
> npm installer sockjs
```

Nous devons ensuite exiger le module sockjs et le module HTTP intégré de Node.
Supprimez la ligne qui nécessite ws et remplacez-la par ce qui suit :

```
const http = require('http');
const sockjs = require('sockjs');
```

Nous modifions ensuite la déclaration de wss pour qu'elle devienne :

```
const wss = sockjs.createServer();
```

Tout en bas du fichier API/index.js, nous allons créer le serveur HTTPS et ajouter les gestionnaires HTTP SockJS :

```
const serveur = http.createServer();
wss.installHandlers(serveur, {préfixe : '/ws'});
serveur.listen(7071, '0.0.0.0');
```

Nous mappons les gestionnaires à un préfixe fourni dans un objet de configuration ('/ws'). Nous demandons au serveur HTTP d'écouter sur le port 7071 (choisi arbitrairement) sur toutes les interfaces réseau de la machine.

La tâche finale consiste à mettre à jour les noms des événements pour qu'ils fonctionnent avec SockJS :

```
ws.on('message',          deviendra          ws.on('données',
client.send(sortant);     deviendra          client.write(sortant);
```

Et voilà, la démo fonctionnera désormais avec les WebSockets là où ils sont pris en charge ; et là où ils ne le sont pas, elle utilisera l'interrogation longue Comet. Cette dernière option de secours affichera un mouvement de curseur légèrement moins fluide, mais elle est plus fonctionnelle que l'absence de connexion du tout !

Exécution de la démo avec SockJS

Si vous avez suivi le didacticiel, vous pouvez exécuter :

```
> installation npm
> démarrer l'exécution de npm
```

Sinon, vous pouvez cloner une version fonctionnelle de la démo à partir de : https://github.com/ably-labs/partage-de-curseur-websockets/arbre/sockjs.

```
> git clone - b sockjs https://github.com/ably-labs/WebSockets-cursor-sharing.git

> installation npm
> démarrer l'exécution de npm
```

Cette démo comprend deux applications : une application Web que nous servons via Snowpack[36] et un serveur Web Node.js. La tâche de démarrage NPM lance à la fois l'API et le serveur Web.

La démo devrait ressembler à celle illustrée ci-dessous :

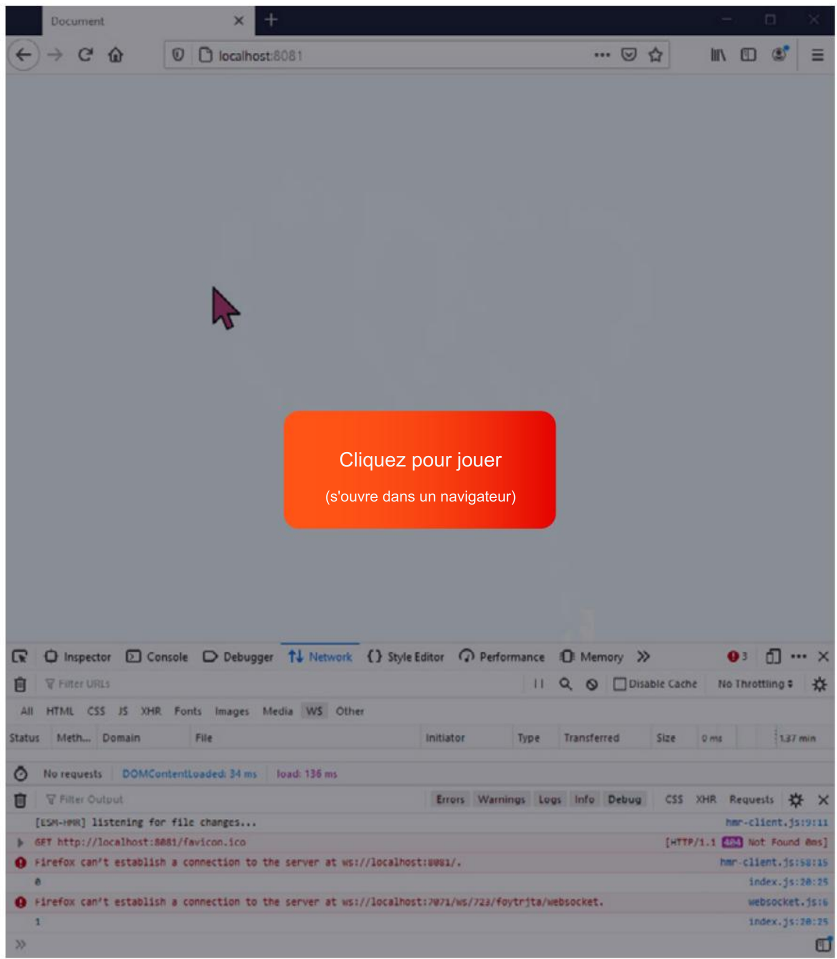

Figure 4.3 : Mouvement du curseur en temps réel grâce à la bibliothèque WebSockets de SockJS

[36] Manteau neigeux

Mise à l'échelle de l'application Web

Vous remarquerez peut-être que dans les deux exemples, nous stockons l'état dans le serveur WebSocket Node.js. Il existe une carte qui garde la trace des WebSockets connectés et de leurs métadonnées associées. Cela signifie que pour que la solution fonctionne et que chaque utilisateur puisse se voir, ils doivent être connectés au même serveur WebSocket.

Le nombre d'utilisateurs actifs que vous pouvez prendre en charge est donc directement lié à la quantité de matériel dont dispose votre serveur. Node.js est assez efficace pour gérer la concurrence, mais une fois que vous atteignez quelques centaines à quelques milliers d'utilisateurs, vous devrez faire évoluer votre matériel pour garder tous les utilisateurs synchronisés.

La mise à l'échelle verticale est souvent une proposition coûteuse, et vous serez toujours confronté à un plafond de performances du matériel le plus puissant que vous pouvez vous procurer. De plus, la mise à l'échelle verticale n'est pas élastique, vous devez donc la faire à l'avance. Vous devez envisager la mise à l'échelle horizontale, qui est meilleure à long terme, mais aussi beaucoup plus difficile. Pour plus de détails, consultez la section L'évolutivité de votre couche serveur dans le chapitre suivant.

Qu’est-ce qui rend les WebSockets difficiles à mettre à l’échelle ?

Les WebSockets sont fondamentalement difficiles à mettre à l'échelle, car les connexions à votre serveur WebSocket doivent être persistantes. Et même une fois que vous avez mis à l'échelle votre couche serveur, vous devez également fournir une solution pour le partage des données entre les nœuds. L'état de connexion doit être stocké hors processus . Cela implique généralement l'utilisation d'un outil tel que Redis37 ou d'une base de données traditionnelle pour garantir que tous les nœuds ont la même vue d'état.

En plus de devoir partager l'état à l'aide d'une technologie supplémentaire, la diffusion vers tous les clients abonnés devient difficile, car chaque nœud WebSocketServer donné ne connaît que les clients qui lui sont connectés.

Il existe plusieurs façons de résoudre ce problème : soit en utilisant une forme de connexion directe entre les nœuds du cluster qui gèrent le trafic, soit en utilisant un mécanisme de publication/abonnement externe. C'est ce que l'on appelle parfois « l'ajout d'un fond de panier » à votre infrastructure, et c'est un autre élément mobile qui rend difficile la mise à l'échelle des WebSockets.

Consultez le chapitre 5 : WebSockets à grande échelle pour une lecture plus approfondie des défis d’ingénierie impliqués dans la mise à l’échelle des WebSockets.

[37] Redis

[38] Tout ce que vous devez savoir sur la publication/l'abonnement

CHAPITRE 5

WebSockets à grande échelle

Ce chapitre aborde les principaux aspects à prendre en compte lorsque vous envisagez de créer un système à grande échelle. J'entends par là un système capable de gérer des milliers, voire des millions, d'appareils d'utilisateurs finaux simultanés lorsqu'ils se connectent, consomment et envoient des messages via WebSockets. Comme vous le verrez, la mise à l'échelle de WebSockets n'est pas anodine et implique de nombreuses décisions d'ingénierie et compromis techniques.

L'évolutivité de votre couche serveur

Il existe deux chemins principaux que vous pouvez emprunter pour faire évoluer votre couche serveur :

- Mise à l'échelle verticale. Également connue sous le nom de mise à l'échelle, elle ajoute plus de puissance (par exemple, CPU, RAM) à un machine existante.
- Mise à l'échelle horizontale. Également connue sous le nom de mise à l'échelle, elle consiste à ajouter plus de machines le réseau, qui partage la charge de travail de traitement.

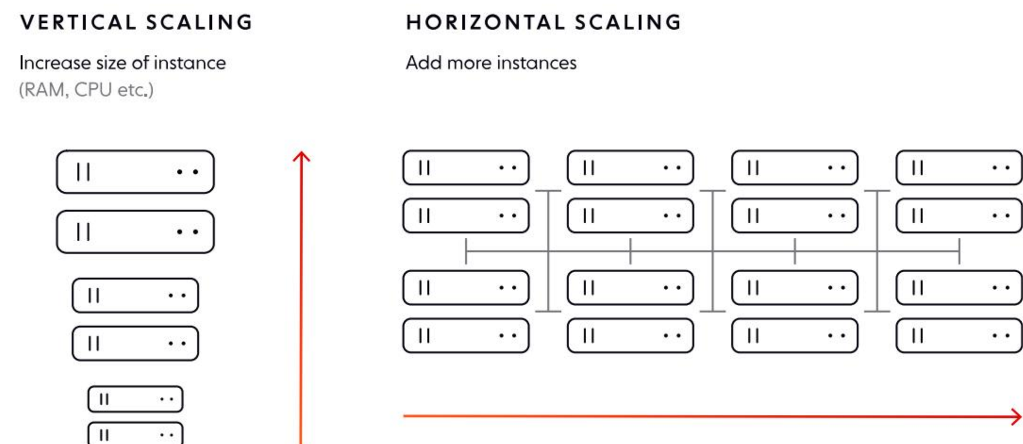

Figure 5.1 : Mise à l'échelle verticale et horizontale

À première vue, la mise à l'échelle verticale semble attrayante, car elle est plus facile à mettre en œuvre et à maintenir que la mise à l'échelle horizontale. Vous pourriez même vous demander : combien de connexions WebSocket un serveur peut-il gérer ? Cependant, c'est rarement la bonne question à poser, et c'est parce que la mise à l'échelle verticale présente de sérieuses limitations pratiques.

Prenons un exemple hypothétique pour illustrer ces inconvénients. Imaginez que vous avez développé une plateforme d'événements virtuels interactifs avec WebSockets qui est utilisée par des dizaines de milliers d'utilisateurs, avec un nombre croissant de nouveaux utilisateurs chaque jour. Cela se traduit par un nombre toujours croissant de connexions WebSocket que votre couche serveur doit gérer.

Cependant, comme vous n'utilisez qu'une seule machine, vous ne pouvez y ajouter qu'une quantité limitée de ressources, ce qui signifie que vous ne pouvez faire évoluer votre serveur que jusqu'à une capacité limitée. De plus, que se passe-t-il si, à un moment donné, le nombre de connexions WebSocket simultanées s'avère trop important pour une seule machine ? Ou que se passe-t-il si vous devez mettre à niveau votre serveur ? Avec la mise à l'échelle verticale, vous disposez d'un point de défaillance unique, ce qui affecterait gravement la disponibilité de votre système et la disponibilité de votre plateforme d'événements virtuels.

> En revanche, la mise à l'échelle horizontale est un modèle plus disponible à long terme. Même si un serveur tombe en panne ou doit être mis à niveau, vous êtes dans une bien meilleure position pour protéger la disponibilité globale de votre système puisque la charge de travail de la machine défaillante est distribuée aux autres nœuds du réseau.

Bien entendu, la mise à l’échelle horizontale comporte ses propres complications : elle implique une architecture plus complexe, une infrastructure supplémentaire à gérer, un équilibrage de charge et une synchronisation automatique des données de message et de l’état de connexion sur plusieurs serveurs WebSocket en quelques millisecondes (plus d’informations sur ces sujets sont abordées plus loin dans ce chapitre).

Malgré sa complexité accrue, la mise à l'échelle horizontale mérite d'être envisagée, car elle permet une mise à l'échelle illimitée (en théorie). Cela en fait une alternative supérieure à la mise à l'échelle verticale. Ainsi, au lieu de se demander combien de connexions un serveur peut gérer, une meilleure question serait : sur combien de serveurs puis-je répartir la charge de travail ?

> Outre la mise à l'échelle horizontale, vous devez également tenir compte de l'élasticité de votre couche serveur. Des complications catastrophiques peuvent survenir lorsque vous exposez une couche serveur inélastique à l'Internet public, une source de trafic volatile et imprévisible. Pour gérer avec succès les WebSockets à grande échelle, vous devez être en mesure d' ajouter dynamiquement (automatiquement) davantage de serveurs au mix afin que votre système puisse s'adapter rapidement et gérer les pics d'utilisation potentiels à tout moment.

Équilibrage de charge

L'équilibrage de charge est le processus de distribution du trafic réseau entrant (connexions WebSocket dans notre cas) sur un groupe de serveurs principaux (généralement appelés batterie de serveurs).
Lorsque vous évoluez horizontalement, votre stratégie d'équilibrage de charge est fondamentale.

Un équilibreur de charge, qui peut être un périphérique physique, une instance virtualisée exécutée sur un matériel spécialisé ou un processus logiciel, agit comme un « agent de la circulation ». Placé entre les clients et votre parc de serveurs back-end, l'équilibreur de charge reçoit puis achemine les connexions entrantes vers les serveurs disponibles capables de les gérer.

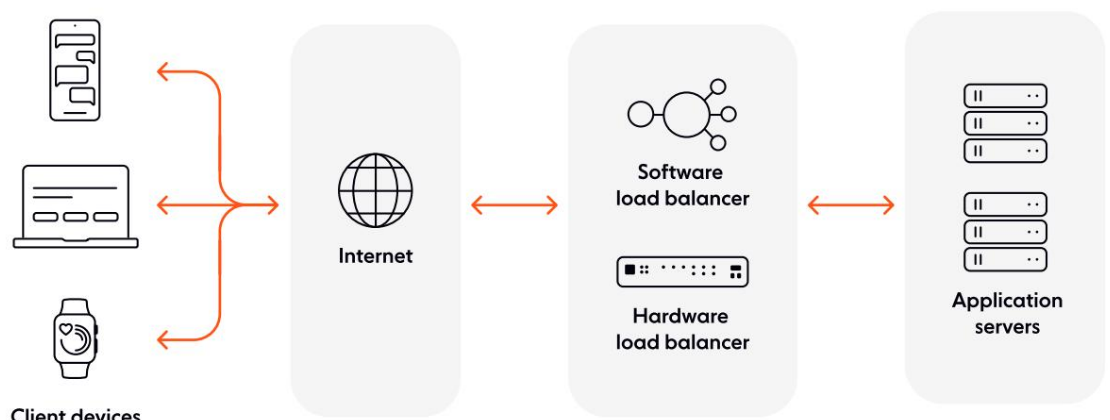

Figure 5.2 : Équilibrage de charge

Les équilibreurs de charge détectent l'état des ressources du backend et n'envoient pas de trafic aux serveurs qui ne peut pas gérer une charge supplémentaire. Si un serveur tombe en panne, l'équilibreur de charge redirige son trafic vers les serveurs opérationnels restants. Lorsqu'un nouveau serveur est ajouté à la batterie de serveurs, l'équilibreur de charge commence automatiquement à lui distribuer le trafic.

L'objectif d'une stratégie d'équilibrage de charge efficace est de :

- Assurer la tolérance aux pannes, la haute disponibilité et la fiabilité.
- Assurez-vous qu'aucun serveur n'est surchargé, ce qui peut dégrader les performances.
- Minimisez le temps de réponse du serveur et maximisez le débit.
- Vous permet d'ajouter ou de supprimer des serveurs de manière flexible, selon la demande.

Vous pouvez effectuer un équilibrage de charge à différents niveaux de l'interconnexion des systèmes ouverts (OSI) Modèle 39 :

- Couche 4 (L4). Équilibrage de charge au niveau du transport. Cela signifie que les équilibreurs de charge peuvent prendre des décisions de routage en fonction des ports TCP ou UDP utilisés par les paquets, ainsi que de leurs adresses IP source et de destination. Le contenu des paquets eux-mêmes n'est pas inspecté.

39 Modèle OSI, Wikipédia

Couche 7 (L7). Équilibrage de charge au niveau de l'application. À ce niveau, les équilibreurs de charge peuvent évaluer une gamme de données plus large qu'au niveau L4, notamment les en-têtes HTTP et les identifiants de session SSL. L'équilibrage de charge L7 est généralement plus sophistiqué et plus gourmand en ressources, mais il peut également être plus efficace en permettant à l'équilibreur de charge de prendre des décisions de routage en fonction du contenu du message.

> L'équilibrage de charge L4 et L7 sont couramment utilisés dans les architectures modernes. L'équilibrage de charge de couche 4 est idéal pour un équilibrage de charge simple au niveau des paquets, et il est généralement plus rapide et plus sécurisé (car les données des messages ne sont pas inspectées). En comparaison, l'équilibrage de charge de couche 7 est plus coûteux, mais il est capable d'un routage intelligent basé sur l'URL (ce que vous ne pouvez pas faire avec l'équilibrage de charge L4). Les grands systèmes distribués utilisent souvent une architecture d'équilibrage de charge L4/L7 à deux niveaux pour le trafic Internet.

Algorithmes d'équilibrage de charge

Un équilibreur de charge suivra un algorithme pour déterminer comment distribuer les requêtes sur votre parc de serveurs. Il existe différentes options à prendre en compte et à choisir. Le tableau ci-dessous présente certaines des options les plus couramment utilisées :

ALGORITHME	À PROPOS
Tournoi à la ronde	Cela implique le routage des connexions vers les serveurs disponibles de manière séquentielle, sur une base cyclique. Pour un exemple simplifié, supposons que nous ayons deux serveurs, A et B. La première connexion va au serveur A, la deuxième va au serveur B, la troisième va au serveur A, la quatrième va au serveur B, et ainsi de suite.
Moins de connexions	Une nouvelle connexion est acheminée vers le serveur avec le moins de connexions actives. relations.
Bande passante minimale	Une nouvelle connexion est acheminée vers le serveur qui dessert actuellement le moins de trafic, mesuré en mégabits par seconde (Mbps).
Temps de réponse minimum	Une nouvelle connexion est acheminée vers la machine qui prend le moins de temps. temps de réponse à une demande de surveillance de l'état de santé (la vitesse de réponse est utilisée pour indiquer le niveau de charge d'un serveur). Certains équilibreurs de charge peuvent également prendre en compte le nombre de connexions actives sur chaque serveur.
Méthodes de hachage	La décision de routage est prise sur la base d'un hachage de divers éléments de données provenant de la connexion entrante. Cela peut inclure des informations telles que le numéro de port, nom de domaine et adresse IP.
Aléatoire avec deux choix	L'équilibreur de charge sélectionne au hasard deux serveurs de votre batterie et achemine un nouvelle connexion à la machine avec le moins de connexions actives.

ALGORITHME	À PROPOS
Chargement personnalisé	L'équilibreur de charge interroge la charge sur les serveurs individuels à l'aide d'un protocole tel que le protocole SNMP (Simple Network Management Protocol)40 et attribue une nouvelle connexion à la machine présentant les meilleures mesures de charge. Vous pouvez définir différents des mesures à surveiller, telles que l'utilisation du processeur, la mémoire et le temps de réponse.

Tableau 5.1 : Algorithmes d'équilibrage de charge

Vous devez sélectionner un algorithme d'équilibrage de charge en fonction des spécificités de votre cas d'utilisation WebSocket. Dans les scénarios où vous avez exactement le même nombre de messages envoyés à tous les clients (par exemple, des mises à jour de score en direct pour tous ceux qui suivent un match de tennis) et où vos serveurs ont des capacités de calcul et de stockage à peu près identiques, vous pouvez utiliser l'approche round robin, qui est plus facile à mettre en œuvre que d'autres alternatives.

Cependant, si, par exemple, vous développez une solution de chat, certaines connexions WebSocket seront plus gourmandes en ressources, car certains utilisateurs finaux sont plus bavards. Dans ce cas, une stratégie de type round robin n'est peut-être pas la meilleure solution, car vous répartiriez en fait la charge de manière inégale sur votre parc de serveurs. Dans un tel contexte, il serait préférable d'utiliser un algorithme comme least

bande passante.

Voici d'autres aspects à garder à l'esprit lorsque vous équilibrez la charge des WebSockets :

Recours aux transports alternatifs

Il est possible que vous rencontriez des situations dans lesquelles vous ne pourrez pas utiliser les WebSockets (par exemple, certains pare-feu et réseaux d'entreprise bloquent les connexions WebSockets). Lorsque cela se produit, votre système doit pouvoir recourir à un autre transport, généralement une technique basée sur HTTP, comme l'interrogation longue Comet. Cela signifie que votre couche serveur doit « comprendre » les deux WebSockets, ainsi que tous les replis que vous utilisez.

Votre couche serveur doit être prête à s'adapter rapidement au retour à un transport moins efficace, ce qui signifie généralement une charge accrue. Vous devez garder à l'esprit que votre stratégie d'équilibrage de charge idéale pour les WebSockets n'est pas toujours la bonne pour les requêtes HTTP ; après tout, les WebSockets avec état et HTTP sans état sont fondamentalement différents.

Lorsque vous concevez votre système, vous devez vous assurer qu'il est capable d'équilibrer avec succès la charge des WebSockets, ainsi que de toutes les solutions de secours HTTP que vous prenez en charge (vous souhaiterez peut-être avoir des

[40] Protocole simple de gestion de réseau, Wikipédia

Sessions collantes

On pourrait soutenir que les WebSockets sont sticky par défaut (dans le sens où il existe une connexion persistante entre le serveur et le client). Cependant, cela ne signifie pas que vous êtes obligé d'utiliser l'équilibrage de charge sticky (où l'équilibreur de charge achemine à plusieurs reprises le trafic d'un client vers le même serveur de destination). En fait, l'équilibrage de charge sticky est une approche plutôt fragile (il y a toujours un risque qu'un serveur tombe en panne), ce qui rend difficile le rééquilibrage de la charge. Plutôt que d'utiliser l'équilibrage de charge sticky, qui suppose par nature qu'un client restera toujours connecté au même serveur, il est plus fiable d'utiliser des sessions non sticky et de disposer d'un mécanisme qui permet à vos serveurs de partager l'état de connexion entre eux. De cette façon, la continuité du flux peut être assurée sans qu'il soit nécessaire qu'une connexion WebSocket soit toujours liée exactement au même serveur.

Concevoir votre système pour l'évolutivité

Lorsque vous créez des applications avec WebSockets pour les utilisateurs finaux se connectant via l'Internet public, vous ne pourrez souvent pas prédire le nombre d'appareils connectés simultanément. Vous devez concevoir votre système de manière à ce qu'il soit capable de gérer un nombre inconnu (mais potentiellement très élevé) et volatil d'utilisateurs simultanés.

> Pour gérer l'imprévisibilité, vous devez concevoir votre système selon un modèle conçu pour une évolutivité considérable. L'un des choix les plus populaires et les plus fiables est le modèle pub/sub[41].

En un mot, pub/sub fournit un cadre pour l'échange de messages entre les éditeurs (généralement votre serveur) et les abonnés (souvent, les appareils des utilisateurs finaux). Les éditeurs et les abonnés ne se connaissent pas, car ils sont découplés par un courtier de messages, qui regroupe généralement les messages en canaux (ou sujets). Les éditeurs envoient des messages aux canaux, tandis que les abonnés reçoivent des messages en s'abonnant aux canaux pertinents.

[41] Tout ce que vous devez savoir sur la publication/l'abonnement

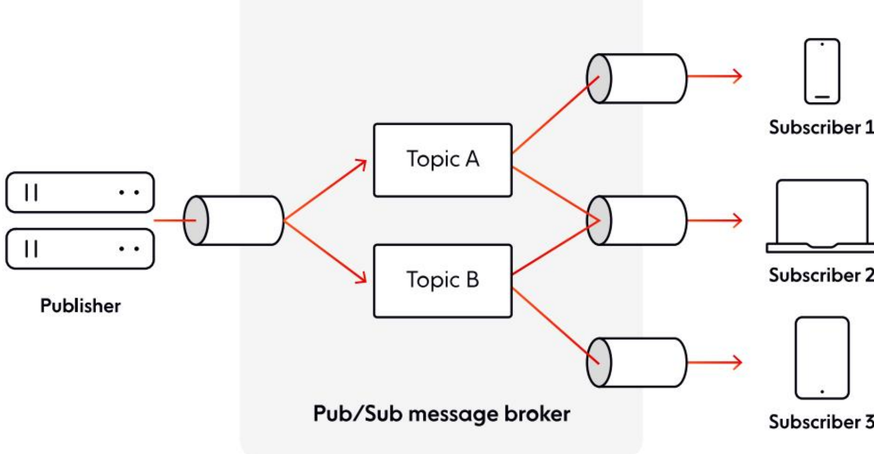

Figure 5.3 : Le modèle pub/sub

La nature découplée du modèle pub/sub signifie que vos applications peuvent théoriquement évoluer vers un nombre illimité d'abonnés. L'un des principaux avantages de l'adoption du modèle pub/sub est que vous n'avez souvent qu'un seul composant qui doit gérer la mise à l'échelle des connexions WebSocket : le courtier de messages. Tant que le courtier de messages peut évoluer de manière prévisible et fiable, il est peu probable que vous ayez à ajouter des composants supplémentaires ou à apporter d'autres modifications à votre système pour gérer le nombre imprévisible d'utilisateurs simultanés se connectant via WebSockets.

Voici quelques autres avantages dont vous bénéficiez en utilisant pub/sub :

- Évolutivité plus fluide. Les systèmes utilisant pub/sub sont évolutifs sans crainte de interrompre la fonctionnalité car la logique de communication et la logique métier sont des entités distinctes. Les architectes logiciels peuvent repenser l'architecture du canal du courtier de messages sans craindre de rompre la logique métier.
- Élasticité. Il n'est pas nécessaire de prédéfinir un nombre défini d'éditeurs ou d'abonnés. Ils peuvent être ajoutés à un canal requis en fonction de l'utilisation.
- Facilité de développement et intégration rapide. Pub/Sub est indépendant de la programmation Protocole de langage et de communication qui permet d'intégrer plus rapidement des composants disparates d'un système par rapport aux alternatives existantes.

Il existe de nombreux projets construits avec WebSockets et pub/sub[42], ainsi que de nombreuses bibliothèques open source et solutions commerciales combinant ces deux éléments. Il est donc peu probable que vous ayez à créer votre propre capacité WebSockets + pub/sub à partir de zéro. Voici quelques exemples de solutions open source que vous pouvez utiliser : Socket.IO avec l'adaptateur pub/sub Redis[43], SocketCluster[44] ou Django Channels[45]. Bien entendu, lorsque vous choisissez une solution open source, vous devez la déployer, la gérer et la faire évoluer vous-même. Il s'agit sans aucun doute d'un défi technique difficile.

42 projets open source Websocket Pubsub sur GitHub

Adaptateur Redis 43 pour Socket.IO

Cluster de sockets 44

45 Canaux Django

Transports de repli

Bien que bénéficiant d'un support de plateforme étendu, les WebSockets souffrent de certains problèmes de réseau. Voici quelques-uns des problèmes que vous pouvez rencontrer :

- Certains proxys ne prennent pas en charge le protocole WebSocket ou mettent fin à la persistance relations.
- Certains pare-feu, VPN et réseaux d'entreprise bloquent des ports spécifiques, tels que 443 (le port d'accès Web standard qui prend en charge les connexions WebSocket sécurisées).
- Les WebSockets ne sont toujours pas entièrement pris en charge par tous les navigateurs.

Imaginez que vous avez développé une plateforme CRM, marketing et commerciale pour des dizaines de milliers d'utilisateurs professionnels, sur laquelle vous disposez de fonctionnalités en temps réel telles que le chat et les tableaux de bord en direct, alimentés par WebSockets. Mais certains utilisateurs peuvent se connecter à partir de réseaux d'entreprise restrictifs qui bloquent ou interrompent les connexions WebSockets. Alors, que faire pour garantir que votre produit est disponible pour vos clients, sachant que vous ne pourrez peut-être pas utiliser WebSockets dans toutes les situations ?

> Si vous prévoyez que des clients se connectent à partir de pare-feu d'entreprise ou de sources autrement délicates, vous devez envisager de prendre en charge les transports de secours.

La plupart des solutions WebSocket intègrent un support de secours. Par exemple, Socket.IO[46], l'une des bibliothèques WebSocket open source les plus populaires, essaiera de manière opaque d'établir une connexion WebSocket si possible, et reviendra à l'interrogation longue HTTP dans le cas contraire.

Un autre exemple est SockJS[47], qui prend en charge un grand nombre de solutions de secours en matière de streaming et d'interrogation, notamment l'interrogation xhr (interrogation longue utilisant XHR interdomaine[48]) et eventsource (événements envoyés par le serveur[49]).

> Consultez le chapitre 4 : Création d'une application Web avec WebSockets pour plus de détails sur la création d'une application en temps réel avec SockJS qui recourt à l'interrogation longue Comet lorsque WebSockets ne peut pas être utilisé.

46 Socket.IO

47 Client SockJS

48 XMLHttpRequest Norme de vie

49 événements envoyés par le serveur (SSE) : une analyse conceptuelle approfondie

Au-delà des solutions open source, la plupart des fournisseurs de solutions WebSocket commerciales prennent également en charge les transports de secours. Bien entendu, il existe également la possibilité de développer votre propre capacité de secours, mais il s'agit d'une tâche complexe et chronophage. Dans la plupart des cas, pour réduire au minimum la complexité technique, il est préférable d'utiliser une solution existante basée sur WebSocket qui inclut des options de secours.

> Dans le contexte de l’échelle, il est essentiel de prendre en compte l’impact que les replis peuvent avoir sur la disponibilité de votre système.

Supposons que vous ayez des milliers, voire des dizaines de milliers, d'utilisateurs connectés simultanément à votre système et qu'un incident se produise qui entraîne le retour d'une proportion importante des connexions WebSocket à une interrogation longue. Non seulement la gestion de dizaines de milliers de connexions WebSocket simultanées est un défi en soi, mais il est encore amplifié par le retour à une interrogation longue, qui est beaucoup plus exigeante pour votre couche serveur. Par exemple, alors que les WebSockets vous permettent de transmettre des données dès qu'elles sont disponibles via des connexions persistantes, avec une interrogation longue, vous devez mettre les données en mémoire tampon et les conserver quelque part jusqu'à ce que la prochaine requête arrive ; cela nécessite beaucoup plus de ressources (utilisation accrue de la RAM).

> Lorsque des dizaines de milliers de connexions WebSocket se rabattent simultanément sur une interrogation longue (ou tout autre transport similaire), votre problème d'évolutivité peut augmenter d'un ordre de grandeur supplémentaire. Pour garantir la disponibilité et le temps de fonctionnement de votre système, votre couche serveur doit être élastique et avoir une capacité suffisante pour gérer la charge accrue. Vous pouvez également envisager d'utiliser un mécanisme de backoff exponentiel (voir Reconnexions automatiques pour plus de détails).

Gestion des connexions et des messages WebSocket

Nous allons maintenant examiner les principaux éléments à prendre en compte lors de la gestion du trafic WebSocket (connexions et messages).

Nouvelles connexions

L'établissement d'une nouvelle connexion WebSocket entraîne une surcharge modérée : le processus implique une paire demande/réponse non triviale entre le client et le serveur, connue sous le nom de poignée de main d'ouverture.

Imaginez maintenant que vous diffusez des mises à jour sportives en direct et qu'un événement très populaire se déroule, comme un match de la Coupe du monde ou une finale du Grand Chelem. Vous pouvez avoir des dizaines de milliers, voire des millions, d'appareils clients essayant d'ouvrir des connexions WebSocket en même temps. Un tel scénario conduit à une énorme explosion du trafic, et votre système doit être prêt à le gérer. La situation serait encore plus compliquée si toutes ces connexions WebSocket devaient simultanément revenir à un transport moins efficace.

Voici quelques mesures que vous pouvez prendre pour vous préparer aux cas où vous devez gérer un nombre extrêmement élevé de connexions WebSocket s'ouvrant simultanément :

- Tests et surveillance. Exécutez des tests de charge et de stress pour évaluer le comportement de votre système en cas de charge de pointe et disposez de mécanismes complets de surveillance et d'alerte en temps réel pour bien comprendre ce qui se passe à tout moment et pouvoir agir immédiatement en cas de problème.

- Appliquer des limites. En fonction des résultats des tests de charge et de stress, vous pouvez appliquer des limites strictes, telles que le nombre maximal de connexions ou le nombre de nouvelles connexions pouvant être ouvertes dans un intervalle de temps spécifique. De cette façon, vous bénéficiez d'une plus grande prévisibilité et êtes mieux placé pour faire évoluer votre système de manière fiable.

- Assurez-vous que votre système est hautement disponible et tolérant aux pannes. Vous devez être en mesure de mettre à l'échelle rapidement (automatiquement) votre couche serveur afin qu'elle puisse gérer les pics de trafic. De plus, il est conseillé de fonctionner avec une certaine marge de capacité et de disposer de sauvegardes pour divers composants du système, afin de garantir la redondance et d'éliminer les points de défaillance uniques.

Surveillance des WebSockets

L'Internet public est une source de trafic volatile et imprévisible, vous avez donc besoin d'une pile de surveillance et d'alerte robuste et complète pour vous donner un aperçu de votre système et de la manière dont il gère les connexions WebSocket.

Nous n'entrerons pas dans les détails des solutions que vous pouvez utiliser pour créer votre pile de surveillance WebSockets : il existe de nombreuses options parmi lesquelles choisir, notamment des outils open source comme Prometheus50 et Grafana51.

Voici quelques-unes des mesures généralement surveillées :

- Nombre de connexions
- Taux de désabonnement
- Nombre de messages, débit et trafic réseau
- Utilisation de la mémoire/du processeur
- Nombre d'instances
- Perte, duplication et réorganisation de paquets
- Latence
- Avertissements et erreurs
- Santé du serveur, du centre de données et de la région

Il est préférable que les indicateurs que vous surveillez soient présentés sur des tableaux de bord en temps réel , afin de toujours avoir une visibilité actualisée sur ce qui se passe. Et, bien sûr, vous devez configurer des alertes pour que, lorsque certains indicateurs sortent des valeurs acceptables, vous puissiez être instantanément averti et réagir rapidement pour résoudre les problèmes potentiels.

Délestage de charge

Lors de la mise à l'échelle de WebSockets, vous devrez inévitablement faire face à la congestion du trafic et au risque que votre couche serveur soit surchargée par le nombre de connexions qu'elle doit gérer. Si la situation n'est pas maîtrisée, elle peut entraîner des pannes en cascade, voire un effondrement total de votre système.

Pour éviter que cela ne se produise, vous devez mettre en place une stratégie de délestage. Les mécanismes de délestage de charge vous permettent généralement de détecter la congestion et d'échouer en douceur lorsqu'un serveur approche de la surcharge, en rejetant une partie ou la totalité du trafic entrant.

50 Prométhée
51 Grafana

Voici quelques éléments à garder à l'esprit lorsque vous supprimez des connexions :

- Vous devez exécuter des tests pour découvrir la charge maximale que votre système supporte généralement capable de gérer. Tout ce qui dépasse ce seuil devrait être un candidat à l'excrétion.
- Vous avez besoin d'un mécanisme de back-off (voir Reconnexions automatiques pour plus de détails) pour empêcher les clients rejetés de tenter de se reconnecter immédiatement ; cela ne ferait que mettre votre système sous plus de pression.
- Vous pouvez également envisager de supprimer les connexions existantes pour réduire la charge sur votre système ; par exemple, celles qui sont inactives (qui, même si elles sont inactives, consomment toujours des ressources en raison des pulsations).

Rétablir les connexions

De nombreuses raisons peuvent entraîner la perte de connexions WebSocket. Les utilisateurs peuvent passer d'un réseau de données mobiles à un réseau Wi-Fi, passer par un tunnel ou rencontrer des problèmes de réseau intermittents. L'un de vos serveurs peut également être surchargé et tomber en panne, ou il doit supprimer des connexions. Lorsque de tels scénarios se produisent, WebSocket les connexions doivent être rétablies.

Reconnexions automatiques

Vous pouvez implémenter un script de reconnexion qui permet aux clients de se reconnecter automatiquement. Un script simple pourrait ressembler à ceci[52] :

```
fonction connecter() {
    ws = nouveau WebSocket("ws://localhost:8080");
    ws.addEventListener('fermer', connecter);
}
```

Cependant, cette approche n'est pas idéale, car les tentatives de reconnexion se produisent immédiatement après la fermeture des connexions WebSocket. Les clients tentent continuellement de se reconnecter, même si votre couche serveur n'a pas la capacité suffisante pour gérer toutes les connexions WebSocket entrantes. Cela peut mettre votre système sous une pression encore plus forte et entraîner des échecs en cascade.

52 Jeroen de Kok, Comment implémenter un algorithme de backoff exponentiel aléatoire en Javascript

Une amélioration serait d'utiliser un algorithme de reconnexion à recul exponentiel, comme le montre cet exemple :

```
var initialReconnectDelay = 1000;
var currentReconnectDelay = initialReconnectDelay;
var maxReconnectDelay = 16000;

fonction connecter() {
	ws = nouveau WebSocket("ws://localhost:8080");
	ws.addEventListener('open', onWebsocketOpen);
	ws.addEventListener('close', onWebsocketClose);
}

fonction onWebsocketOpen() {
	délai de reconnexion actuel = délai de reconnexion initial;
}

fonction onWebsocketClose() {
	ws = nul;
	setTimeout(() => {
		reconnecterToWebsocket();
	}, délai de reconnexion actuel);
}

fonction reconnectToWebsocket() {
	si (délai de reconnexion actuel < délai de reconnexion maximal) {
		currentReconnectDelay*=2;
	}
	connecter();
}
```

L'algorithme augmente de manière exponentielle le délai après chaque tentative de reconnexion, augmentant le temps d'attente entre les tentatives jusqu'à un délai de reconnexion maximal. Par rapport à un simple script de reconnexion, c'est mieux, car cela vous donne le temps d'ajouter plus de capacité à votre système afin qu'il puisse gérer toutes les reconnexions WebSocket. Mais ce n'est toujours pas génial, car tous les clients continueraient à essayer de se reconnecter en même temps.

Vous pouvez rendre le mécanisme de backoff exponentiel plus fiable en le rendant aléatoire, de sorte que tous les clients ne se reconnectent pas exactement au même moment :

```
fonction onWebsocketClose() {
	ws = nul;
	// Ajoutez une valeur comprise entre 0 et 3000 ms au délai.
	setTimeout(() => {
		reconnecterToWebsocket();
	}, currentReconnectDelay + Math.floor(Math.random() * 3000)));
}
```

Reconnexions avec la continuité

Dans certains cas d'utilisation, l'intégrité des données (ordre garanti et livraison exacte) est cruciale, et une fois la connexion WebSocket rétablie, le flux de données doit reprendre exactement là où il s'était arrêté. Pensez, par exemple, à des fonctionnalités comme le chat en direct, où l'absence de messages en raison d'une déconnexion ou leur réception dans le désordre entraîne une mauvaise expérience utilisateur et provoque confusion et frustration.

Si reprendre un flux exactement là où il s'était arrêté après de brèves déconnexions est important pour votre cas d'utilisation, voici quelques éléments que vous devrez prendre en compte :

- Mise en cache des messages dans la mémoire frontale. Combien de messages stockez-vous et pour combien de temps ? combien de temps?
- Déplacement de données vers un stockage persistant. Avez-vous besoin de transférer des données vers un stockage persistant Stockage ? Si oui, où les stockez-vous et pendant combien de temps ? Comment les clients accéderont-ils à ces données lorsqu'ils se reconnecteront ?
- Comment le flux reprend-il ? Lorsqu'un client se reconnecte, comment savez-vous exactement Où reprendre le flux ? Devez-vous utiliser un numéro de série/un horodatage pour déterminer où une connexion a été interrompue ? Qui doit suivre la rupture de la connexion : le client ou le serveur ?
- Synchronisation de l'état de connexion sur vos serveurs. En supposant que vous n'utilisiez pas l'équilibrage de charge permanent (ce qui ne devrait pas être le cas), un client peut se reconnecter à un serveur différent du serveur initial. Vous devez donc vous assurer que n'importe quel serveur est en mesure de reprendre le flux. Devez-vous utiliser un protocole de partage de trafic ou une solution de publication/abonnement pour garantir que l'état de connexion est partagé sur l'ensemble de votre parc de serveurs ? Comment allez-vous vous assurer que le mécanisme de synchronisation lui-même est toujours disponible et fonctionne de manière fiable ?

Pulsations cardiaques

Le protocole WebSocket prend en charge nativement les trames de contrôle[53] appelées Ping et Pong. Ces trames de contrôle sont un mécanisme de battement de cœur au niveau de l'application pour détecter si une connexion WebSocket est active. En général, c'est le serveur qui envoie une trame Ping et, à la réception, le côté client doit renvoyer une trame Pong en guise de réponse.

> Vous devez surveiller de près l'effet des battements de cœur à grande échelle sur votre système, ainsi que le rapport entre les trames Ping/Pong et les messages réels envoyés via WebSockets. Il existe des situations dans lesquelles vous pouvez constater que vous envoyez plus de battements de cœur que de messages (trames texte ou binaires) via WebSockets.

53 RFC 6455, Section 5.5 : Trames de contrôle

Cela n'a pas vraiment d'impact dans le contexte d'une seule connexion, mais avoir des milliers, voire des millions de connexions WebSocket simultanées avec une fréquence de pulsation élevée, ajoutera une charge importante sur votre serveur. Si votre cas d'utilisation le permet, il peut être judicieux de réduire la fréquence des pulsations pour faciliter la mise à l'échelle.

Contre-pression

Lorsque vous diffusez des données vers des appareils clients à grande échelle via Internet, la contre-pression est l'un des principaux problèmes auxquels vous devrez faire face. Par exemple, supposons que vous diffusez 20 messages par seconde, mais qu'un client ne peut en gérer que 15 par seconde. Que faites-vous des 5 messages restants par seconde que le client ne peut pas consommer ?

Vous devez disposer d'un moyen de surveiller les tampons accumulés sur les sockets utilisés pour diffuser des données aux clients et de garantir qu'un tampon ne dépasse jamais ce que la connexion en aval peut supporter. Au-delà des problèmes côté client, si vous ne gérez pas activement les tampons, vous risquez d'épuiser les ressources de votre couche serveur. Cela peut se produire très rapidement lorsque vous avez des milliers de connexions WebSocket simultanées.

Une mesure corrective typique de contre-pression consiste à supprimer les paquets sans distinction. Cette approche fonctionne bien lorsque le dernier message envoyé à partir d'un flux WebSocket est toujours le plus important, par exemple dans des cas d'utilisation tels que les mises à jour sportives en direct, où le dernier score est l'information la plus pertinente. Pour réduire la bande passante et la latence, en plus de supprimer les paquets, vous devez également envisager une compression delta de message, qui utilise généralement un algorithme de diff[54] pour envoyer uniquement les modifications du message précédent au consommateur plutôt que l'intégralité du message.

Cependant, la suppression de paquets n'est pas toujours une bonne solution : il existe des cas d'utilisation où l'intégrité des données est essentielle et où vous ne pouvez tout simplement pas vous permettre de perdre des informations. Dans de tels scénarios, vous devez utiliser des accusés de réception au niveau de l'application (ACK) comme confirmation de la réception des messages et configurer votre système pour qu'il n'envoie pas de lots de messages supplémentaires jusqu'à ce qu'il ait reçu les ACK. Vous devez également réfléchir à la manière de garantir la continuité du flux même en cas de déconnexion.

54 Tsviatko Yovtchev, Delta Compression : Un guide pratique sur les algorithmes de comparaison et les formats de fichiers delta

Un petit mot sur la tolérance aux pannes

Lorsque vous essayez de créer des applications évolutives et prêtes pour la production avec des WebSockets servant des milliers, voire des millions de consommateurs, vous devez inévitablement penser à la tolérance aux pannes[55] de votre système.

Les conceptions tolérantes aux pannes traitent les pannes comme une routine. Il faut partir du principe que les pannes de composants se produiront tôt ou tard. Plus le système est grand, plus les risques de problèmes sont élevés. Ce qui est important, c'est que, lorsque des pannes se produisent, votre système dispose d'une redondance suffisante pour continuer à fonctionner, avec des fonctionnalités et une expérience utilisateur préservées aussi efficacement que possible.

Sans entrer dans les détails, pour rendre votre système tolérant aux pannes, vous devez vous assurer qu'il est redondant en cas de panne du serveur et du centre de données. Si vous créez des applications basées sur le cloud, cela implique tout d'abord d'avoir la possibilité de faire évoluer de manière élastique votre couche serveur (et de fonctionner avec une capacité supplémentaire en veille) et de distribuer votre infrastructure sur plusieurs zones de disponibilité.

Cependant, il ne suffit pas de s'appuyer sur une région spécifique pour de multiples raisons[56] : parfois, plusieurs zones de disponibilité dans une région échouent en même temps ; parfois, il peut y avoir des problèmes de connectivité locale rendant la région inaccessible ; et parfois, il peut simplement y avoir des limitations de capacité dans une région qui empêchent tous les services d'y être pris en charge.

Il est donc souvent préférable de déployer l'infrastructure dans plusieurs régions. C'est le meilleur moyen de garantir l'indépendance statistique des pannes et la meilleure garantie que votre système continuera à fonctionner et à fournir un service ininterrompu à vos utilisateurs.

55 Dr. Paddy Byers, Ingénierie de la fiabilité et de la tolérance aux pannes dans un système distribué

56 Michael Gariffo, AWS subit la troisième panne du mois

Liste de contrôle des WebSockets à grande échelle

- [x] Utilisez la mise à l'échelle horizontale plutôt que verticale. Elle est plus fiable, en particulier pour les cas d'utilisation où vous ne pouvez pas vous permettre que votre système soit indisponible en toutes circonstances.

- [x] Si possible, utilisez des machines plus petites (serveurs) plutôt que des grandes. Elles sont plus faciles et plus rapides à mettre en place, et les coûts sont plus précis.

- [x] L'objectif est d'avoir une ferme de serveurs homogène. Il est beaucoup plus compliqué d'équilibrer la charge de manière efficace et uniforme entre des machines ayant des configurations différentes.

- [x] Ayez une bonne compréhension de votre cas d'utilisation et des paramètres pertinents (tels que les modèles d'utilisation et la bande passante) avant de choisir un algorithme d'équilibrage de charge.

- [x] Assurez-vous que votre couche serveur est élastique de manière dynamique, afin de pouvoir évoluer rapidement en cas de pics de trafic. Vous devez également fonctionner avec une certaine marge de capacité et disposer de sauvegardes pour divers composants du système, afin de garantir la redondance et d'éliminer les points de défaillance uniques.

- [x] Il est rare qu'un protocole unique soit utilisé dans les systèmes à grande échelle. Certains protocoles servent mieux que d'autres des objectifs différents. Vous devez réfléchir aux autres options que votre système doit prendre en charge en plus des WebSockets et envisager des moyens de garantir l'interopérabilité des protocoles.

- [x] Vous devrez probablement prendre en charge les transports de secours, tels que l'interrogation longue Comet, car les WebSockets, bien que largement pris en charge, sont bloqués par certains pare-feu et réseaux d'entreprise. Notez que le recours à un autre protocole modifie vos paramètres de mise à l'échelle. Après tout, les WebSockets avec état sont fondamentalement différents du HTTP sans état, vous avez donc besoin d'une stratégie pour mettre à l'échelle les deux.

- [x] Exécutez des tests de charge et de stress pour comprendre comment votre système se comporte sous une charge de pointe et appliquez des limites strictes (par exemple, le nombre maximal de connexions WebSocket simultanées) pour avoir une certaine prévisibilité.

- Les connexions WebSocket et le trafic sur l'Internet public sont imprévisibles et évoluent rapidement. Vous avez besoin d'une pile de surveillance et d'alerte en temps réel robuste pour vous permettre de détecter et de mettre en œuvre rapidement des mesures correctives en cas de problème.

- Vous devez avoir une stratégie de délestage ; une défaillance gracieuse est toujours préférable à un effondrement total de votre système.

- Utilisez une infrastructure à plusieurs niveaux pour vous permettre de récupérer après une panne et de coordonner les serveurs.

- Certaines connexions WebSocket seront inévitablement interrompues à un moment donné. Vous avez besoin d'une stratégie pour vous assurer qu'une fois les connexions WebSocket restaurées, vous pouvez reprendre le flux avec la garantie de classement et de livraison (de préférence en une seule fois).

- Utilisez un mécanisme de backoff exponentiel aléatoire lors de la gestion des reconnexions. Cela vous permet de protéger votre couche serveur contre toute surcharge, d'éviter les pannes en cascade et de vous donner le temps d'ajouter plus de capacité à votre système.

- Gardez une trace des connexions inactives et fermez-les. Même si aucun message (texte ou trames binaires) n'est envoyé, vous continuez à envoyer des trames ping/pong périodiquement, donc même les connexions inactives consomment des ressources.

Ressources

Références

• Alex Russell, Comet : Données à faible latence pour le

navigateur • Sockets Berkeley

• Puis-je utiliser les WebSockets ?

• Chaînes Django • Dr.

Paddy Byers, Ingénierie de la fiabilité et de la tolérance aux pannes dans un système distribué • Tout ce que vous devez

savoir sur la publication/l'abonnement

• Grafana

• Registre des numéros de code de fermeture IANA

WebSocket • Registre des noms d'extension IANA

WebSocket • Registre des codes d'opération

IANA WebSocket • Registres des protocoles

IANA WebSocket • Registre des noms de sous-protocoles

IANA WebSocket • Groupe de travail HTTP IETF, Documentation HTTP, Spécifications de

base • Journaux IRC,

18.06.2018 • Jeroen de Kok, Comment implémenter un algorithme de backoff exponentiel aléatoire en Javascript •

Jesse James Garrett, Ajax : une nouvelle approche des applications Web • Kayla

Matthews, MQTT : une analyse conceptuelle approfondie •

Sondage long — Concepts et considérations • Matthew

O'Riordan, Google — des sondages comme dans les années 90 •

Michael Gariffo, AWS subit sa troisième panne du mois

• Modèle OSI

• Prométhée

• Redis

• Adaptateur Redis pour Socket.IO •

RFC 1945 : Protocole de transfert hypertexte - HTTP/1.0 • RFC

2068 : Protocole de transfert hypertexte - HTTP/1.1

• RFC 6455 : Le protocole WebSocket

• RFC 7519 : jeton Web JSON (JWT)

• RFC 8441 : Amorçage de WebSockets avec HTTP/2 • Événements

envoyés par le serveur (SSE) : une analyse conceptuelle

approfondie • Événements envoyés par le serveur,

norme HTML Living • Protocole de gestion de réseau

simple • Snowpack

• Cluster de sockets

• Socket.IO

- Noeud SockJS
- Le groupe de travail HTTP de l'IETF
- Le HTTP original tel que défini en 1991
- Le protocole de messagerie orienté texte simple (STOMP)
- Le framework d'extensions WebSocket
- Tsviatko Yovtchev, Delta Compression : Un guide pratique sur les algorithmes de diff et le fichier delta formats
- Listes de diffusion W3C, commentaires sur TCPConnection
- Projets open source Websocket Pubsub sur GitHub
- Sockets Web, norme HTML vivante
- ws : une bibliothèque WebSocket Node.js
- XMLHttpRequest Norme de vie

Vidéos

- Guide du débutant sur les WebSockets
- Le guide complet des WebSockets
- Cours intensif sur les WebSockets - Poignée de main, cas d'utilisation, avantages et inconvénients et plus encore

Lectures complémentaires

- Sécurité WebSockets : principales attaques et

risques • Sécurité WebSocket - Détournement intersite (CSWSH)

- L'avenir des logiciels Web est HTML sur WebSockets
- Implémentation d'un serveur WebSocket avec Node.js
- Migration de millions de WebSockets simultanés vers Envoy (Slack Engineering)
- Le tableau périodique du temps réel

Bibliothèques WebSocket open source

- Socket.IO
- Nodejs-websocket
- Noeud WebSocket
- Noeud SockJS
- Client SockJS
- ws
- websocket-comme-promis
- nœud websocket-faye
- Socle
- rpc-websockets

Réflexions finales

Nous espérons que ce livre vous a aidé à bien comprendre la création et le fonctionnement des WebSockets, et vous a permis de créer facilement votre première application en temps réel basée sur WebSocket. Dans les versions futures, nous prévoyons d'ajouter davantage d'applications de démonstration et de couvrir d'autres aspects qui sont actuellement hors de portée, tels que la sécurité WebSocket et les alternatives aux WebSockets.

Nous apprécions tous les commentaires de nos lecteurs. Si vous avez repéré une erreur, si vous avez des suggestions sur ce que nous devrions inclure dans les futures versions de l'ebook, ou si vous souhaitez simplement discuter des WebSockets, contactez-nous !

Contactez-nous

À propos d'Ably

Ably est la plateforme qui permet de synchroniser des expériences numériques en temps réel pour des millions d'appareils connectés simultanément dans le monde entier. Qu'il s'agisse d'assister à un événement dans un lieu virtuel, de recevoir des informations financières en temps réel ou de surveiller les données de performances d'une voiture en direct, les consommateurs s'attendent simplement à des expériences numériques en temps réel comme standard.

Ably propose une suite d'API permettant de créer, d'étendre et de proposer des expériences numériques puissantes en temps réel, principalement via WebSockets, pour plus de 250 millions d'appareils dans 80 pays chaque mois. Des organisations comme Bloomberg, HubSpot, Verizon et Hopin dépendent de la plateforme d'Ably pour se décharger de la complexité croissante de la synchronisation des données en temps réel, essentielle à l'entreprise, à l'échelle mondiale.

Inscrivez-vous pour un compte gratuit