Prototype Calculateur National OTP2

Codeurs en Liberté / Andrew Byrd

Résumé

La Direction générale des infrastructures, des transports et de mobilités a commandé en février 2024 la réalisation d'une étude visant à établir la faisabilité du déploiement d'un logiciel libre de calcul d'itinéraires, OpenTripPlanner version 2 à l'échelle de la France en intégrant des données provenant du Point d'Accès National aux données de transport, transport.data.gouv.fr.

Les objectifs de cette étude sont les suivants :

1. Intégration de données des transports en commun et de la voirie ;
2. Observations sur la qualité des sources de données sélectionnées, lacunes, accord entre données théoriques et temps réel ;
3. Test de plusieurs calculs d'itinéraires via API HTTP ;
4. Prérequis et chiffrage en matière d'hébergement, load balancing et DevOps ;
5. Évaluation de la pertinence des logiciels open source existants pour l'interface web.

L'étude réalisée comporte les données suivantes : le réseau routier français (données OpenStreetMap), 34 fichiers GTFS couvrant les 100 communes les plus peuplées de France et les données temps réel associées lorsqu'elles sont disponibles. Elle s'est concrétisée par la mise en oeuvre d'un site web de test.

Ces tests ont permis de formuler des recommandations en matière de dimensionnent d'infrastructure technique : nombre de machines, coeurs, mémoires et coûts associés. L'étude met en évidence les points de vigilance suivants : la correspondance des identifiants entre les données théoriques et en temps réel, la difficulté à collecter des données tarifaires, la gestion des correspondances, le nommage des arrêts et l'utilité d'une base nationale des arrêts.

L'étude a été réalisée par l'entreprise Codeurs en Liberté et Andrew Byrd, consultant OpenTripPlanner.

Table des matières

Objectifs

L'objectif de ce projet est d'évaluer la faisabilité d'un calculateur d'itinéraire à l'échelle géographique de la France métropolitaine basé sur le logiciel OpenTripPlanner. On établira des ordres de grandeur sur le matériel exigé ainsi que les efforts de développement logiciel et opérationnel, dans différentes perspectives d'utilisation et volumes d'utilisateurs.

L'effort se concentrera sur les points suivants :
1) Intégration de données TC, voirie. Observations sur qualité des sources de données sélectionnées, lacunes, accord entre données théoriques et temps réel. La prise en charge de la tarification sera décrite mais sans doute rudimentaire.
2) Calcul d'itinéraire via API HTTP machines locales et en centre de données. Exemples de requêtes et réponses, temps de réponses et débits assurés par matériel divers. Exemples de fichiers de configuration et leur impact sur les performances.
3) Hébergement, load balancing, devops : réutilisation des efforts de conteneurisation existants.
4) Interface web - évaluation de la pertinence des logiciels open source existants.

Le but ultime est de fournir des paramètres et fixer les attentes autour d'un éventuel appel d'offres pour la construction d'un tel système d'information voyageurs.

Ressources externes

Certaines étapes importantes de la préparation et de l'évaluation des serveurs OTP sont documentées dans deux dépôts Git aux adresses suivantes :

https://gitlab.com/CodeursEnLiberte/france-planner-xp/data
https://gitlab.com/CodeursEnLiberte/france-planner-xp/perf

Le dépôt *data* contient un script Python qui télécharge l'ensemble des jeux de données d'entrée TC, puis prépare un fichier de configuration `build-config.json` qui leur fait référence. Ce fichier de configuration dirige OTP à créer sa représentation interne du réseau de transports (le graphe). Avec quelques modifications, ce script pourrait servir de base pour une mise à jour régulière (une voire plusieurs fois par jour) d'un serveur de démonstration.

Le dépôt *perf* contient des fichiers de configuration pour plusieurs outils de test de charge, ainsi que des scripts qui génèrent un ensemble de requêtes aléatoires utilisé par ces tests.

Préparation des données d'entrée

OpenTripPlanner construit sa représentation interne du réseau de transports à partir de données sur la voirie et sur les lignes de transports en commun dans des formats ouverts : OpenStreetMap (OSM) et GTFS ou NeTEx respectivement.

La voirie OpenStreetMap

La base de données OSM est mondiale et unifiée, donc il s'agit d'en extraire une zone géographique, puis d'enlever les objets qui ne servent pas au calcul d'itinéraire pour alléger le fichier et accélérer le processus d'import.
Plusieurs services existent pour extraire des zones géographiques de la base centrale OpenStreetMap. Pour cette étude nous avons employé celui de Geofabrik, une entreprise de Karlsruhe en Allemagne, qui est la référence depuis longtemps dans la matière.
https://download.geofabrik.de/europe/france.html
OpenTripPlanner ne consomme les données OSM qu'au format PBF (binaire, plus compact et plus rapide à lire que XML mais lisible uniquement par machine). NB: Les données pour la France ne couvrent que la France métropolitaine.
Ce fichier est d'environ 4,2 Gio compressé. Il sera décompressé et entièrement chargé en mémoire vive lors de l'import, une opération relativement lente et gourmande en ressources. Or, le nom OpenStreetMap est aujourd'hui inexact car ce n'est qu'une fraction de la base qui est consacrée à la voirie (les *streets*). Les exports consistent surtout des formes du bâti, des cours d'eau, des zones d'utilisation du sol, etc. destinés à la cartographie visuelle. Le fichier peut être filtré avec plusieurs outils spécialisés. Nous avons employé Osmium avec la commande suivante :

```
osmium tags-filter france-latest.osm.pbf w/highway
wa/public_transport=platform wa/railway=platform w/park_ride=yes
r/type=restriction r/type=route -o france-filtered.osm.pbf -f
pbf,add_metadata=false
```

Ainsi le fichier qui représente la France métropolitaine est réduit de 4,2 Gio à 788 Mio, soit une réduction de 81%. OpenTripPlanner peut effectuer ce filtrage en interne, mais seulement après avoir consommé 5 fois plus de mémoire que nécessaire donc il est fortement conseillé de pré-filtrer. C'est une étape nécessaire pour rendre la construction du réseau possible sur du matériel plus courant.

Autrefois cette opération aurait exigé un serveur loué en centre de données, à cause de ses fortes demandes en ressources de calcul, et surtout de mémoire vive et de stockage. En 2024, elle se fait aisément sur une station de travail ou ordinateur portable de dernière génération, à condition d'y avoir installé plus de 30 Gio de mémoire et un dispositif de stockage de type SSD.

Préparation d'une couche voirie sans TC

De façon générale, les données OSM changent plus lentement que les données TC, et les changements ont moins d'impact sur les itinéraires produits. Cependant, le chargement des données OSM est plus gourmand en ressources et en temps de calcul que la préparation des données TC par OpenTripPlanner. Pour répondre à cette situation, OTP permet de construire d'abord un réseau de voirie sans TC et de le stocker, puis de réutiliser ce même réseau de voirie comme couche de base sur laquelle on calque un réseau TC dans une opération plus fréquente et rapide. Par exemple, le réseau OSM peut être rafraîchi une fois par semaine ou une fois par

mois, puis combiné avec un réseau TC reconstruit quotidiennement voire plusieurs fois par jour pour prendre en compte les dernières modifications aux horaires et aux services TC.
Après avoir mis le fichier OSM filtré dans un dossier à part consacré au serveur OTP France, nous avons préparé la couche OSM avec la commande suivante :

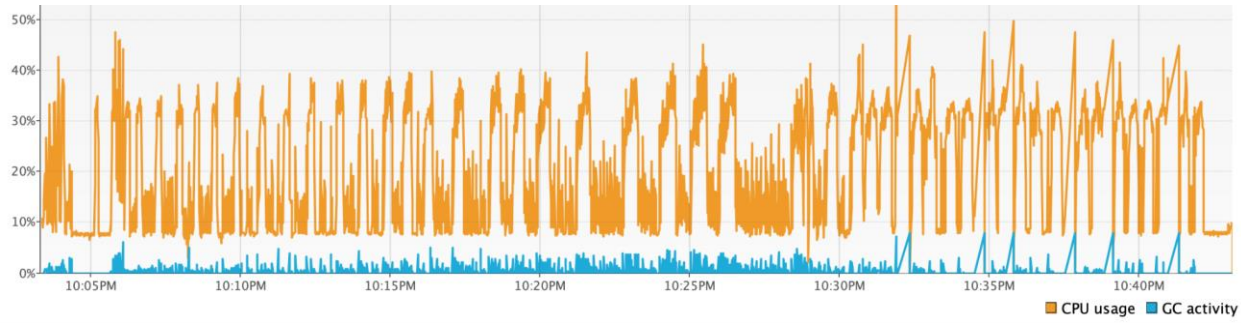
```
java -Xmx30G -jar target/otp-2.5.0-SNAPSHOT-shaded.jar --build --save ~/data/otp/fr
```

Les paramètres par défaut conviennent (areaVisibility désactivé). Il n'y a pas besoin d'un fichier build-config.json pour cette étape.

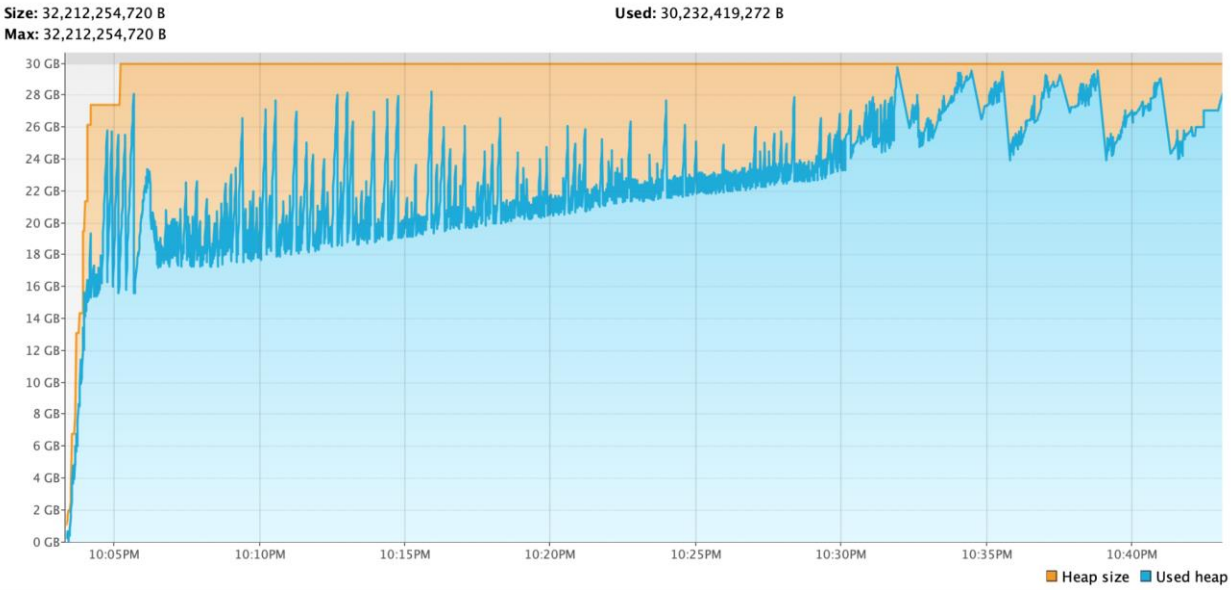
Cette opération prend environ 40 minutes, qui se décomposent de la manière suivante :

- Chargement des données OSM : une minute
- Réparation des erreurs de topologie : 1.5 minutes
- Conversion des voies OSM vers le modèle de graphe interne : 23 minutes
- Indexation de la géométrie et liaison les parkings : 1.5 minutes
- Identification et traitement des sous-graphes isolés : 10 minutes
- Écriture du fichier d'export : 1.5 minutes

Le fichier *graph.obj* produit ainsi qui constitue la représentation interne OTP de la voirie pour la France métropolitaine pèse 2,9 GiO sur disque.
Les 30 Gio de mémoire vive consacrés au tas (*heap* JVM) sont à peine suffisants pour cette étape. Une fois chargée, la représentation des données OSM d'entrée prend une quantité constante et massive de mémoire. S'ajoute à cette consommation de base la mémoire occupée par la représentation interne finale du réseau, qui est bien plus compacte mais qui croît au fur et à mesure que le processus d'ingestion avance (voir fig. 2). On observe vers la fin du processus que la JVM peine à libérer suffisamment de mémoire pour avancer : le temps passé au ramassage (GC) augmente, et le temps passé au travail principal de construction de graphe diminue. Pour assurer la stabilité d'un système d'import automatisé, il faudrait prévoir plus de mémoire vive et l'accorder à la JVM dans la commande de lancement (option -Xmx), ce qui est entièrement réaliste en centre de données, où des instances ayant 64 Gio de mémoire vive sont courantes.



Activité de calcul et de ramassage (GC) lors de la construction du graphe voirie seule.

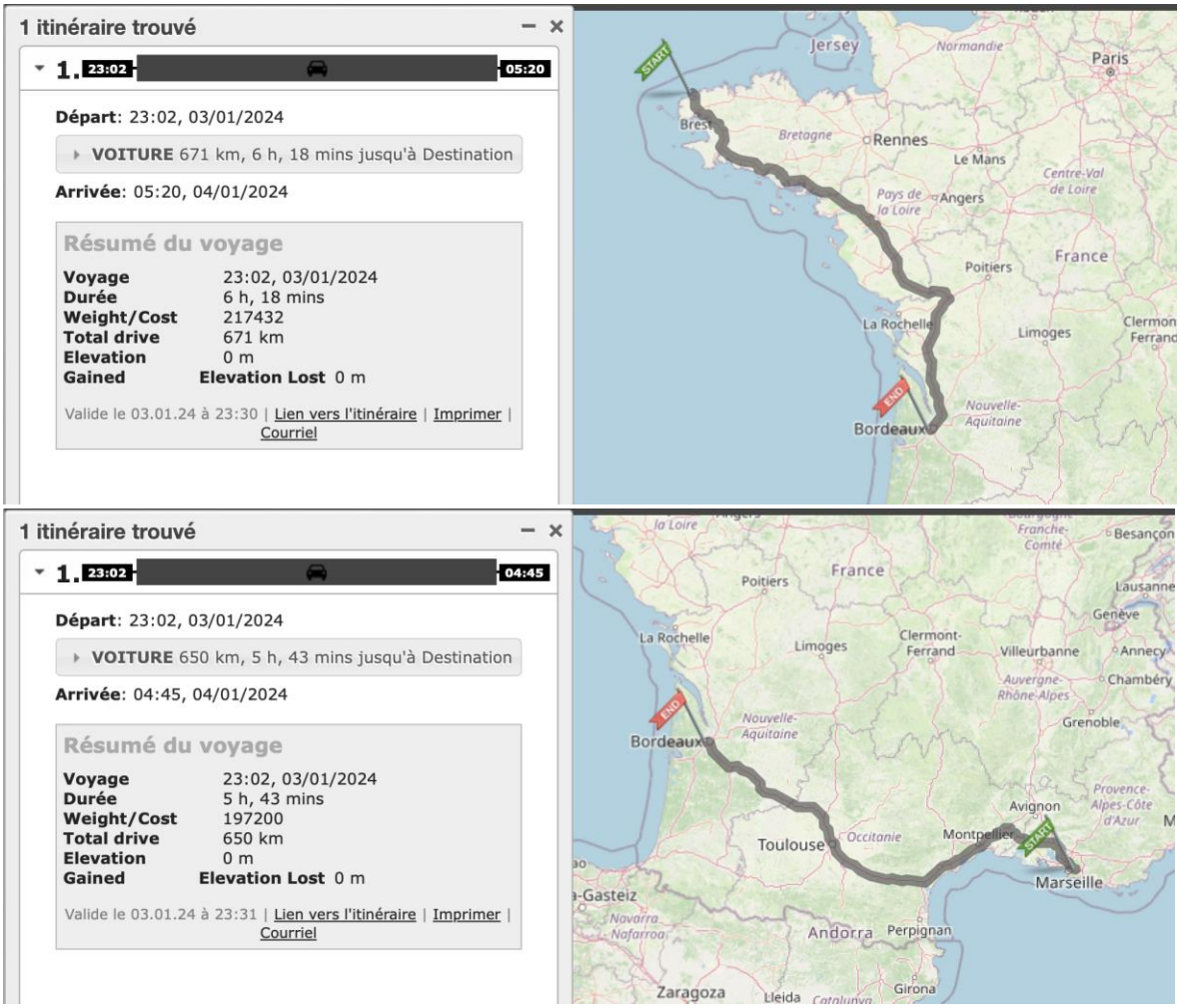


Consommation de mémoire vive lors de la construction du graphe voirie seule.

Nous avons constaté un épuisement de la mémoire vive sur la machine distante en centre de données qui n'a pas eu lieu sur un ordinateur portable MacOS, apparemment grâce à la stratégie de compression de mémoire inactive de ce dernier. Sur un serveur Linux sans zram, swap, ou zswap, réservez quelques Gio supplémentaires en dehors du tas JVM pour le système d'exploitation**.** Une fois le graphe de la voirie construit, les prochaines étapes d'ajout des transports en commun et de calcul d'itinéraires auront des besoins plus modestes en dessous de 32 Gio. Nous avons pu nous débrouiller avec moins de mémoire en ajoutant du swap de façon temporaire, le temps de construire le graphe de la voirie seule, mais le processus était sensiblement ralenti.

À ce stade, il est possible de démarrer un serveur OTP et tester le calcul d'itinéraires sur la voirie seule, sans transports en commun. Le logiciel prend une minute pour charger le fichier de graphe, puis 30 secondes pour l'indexer. Au repos avant d'effectuer un calcul d'itinéraire, il consomme environ 7,5 Gio de mémoire vive. Dans un environnement non-contraint, c'est-à-dire où le tas n'est à aucun moment épuisé pendant le chargement du graphe, il en consomme environ 12-15 Gio. Nous utilisons la commande suivante pour démarrer un serveur qui ouvrira le fichier de réseau produit dans l'étape précédente, puis servir l'API :

```
java -Xmx20G -jar target/otp-2.5.0-SNAPSHOT-shaded.jar --load ~/data/otp/fr --serve
```



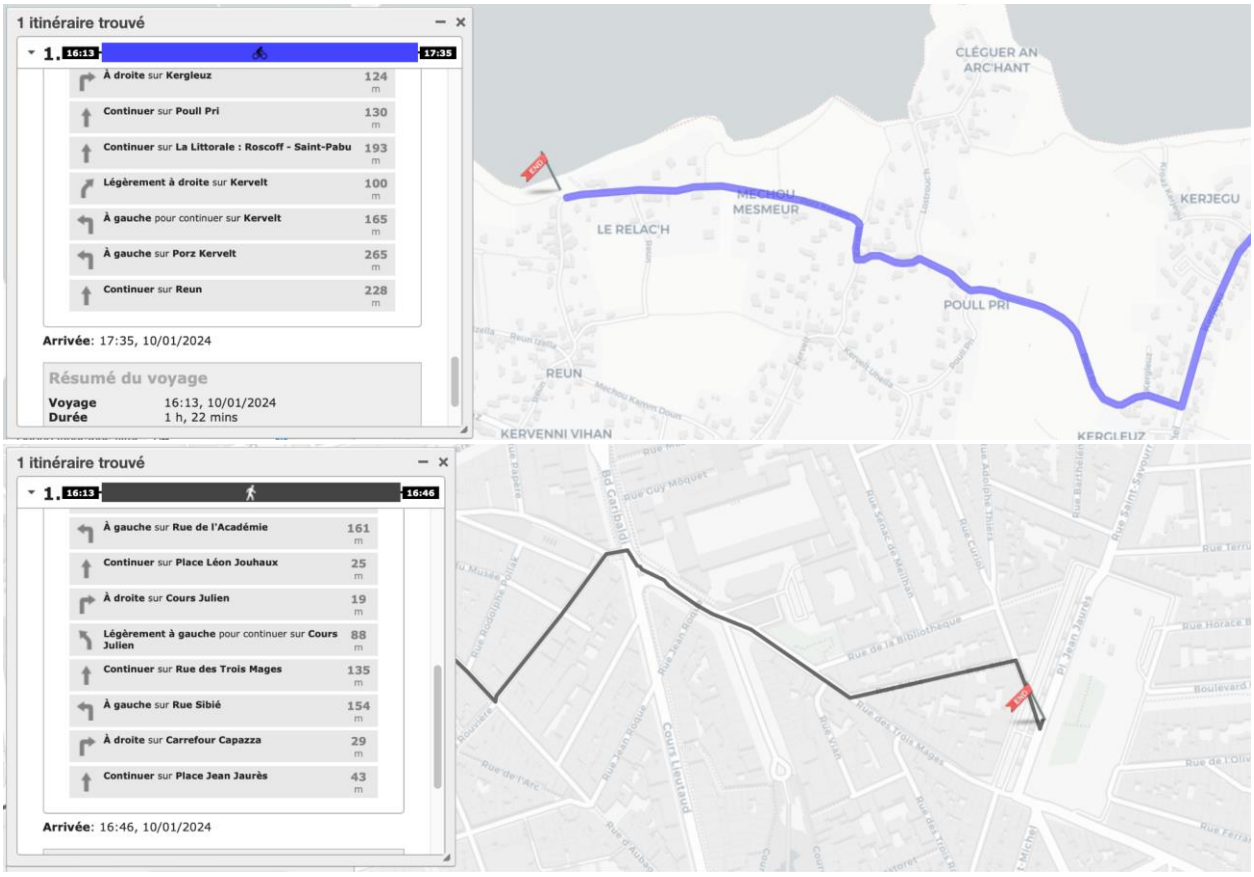
Deux itinéraires longs en voiture sur un seul graphe.

Il est possible d'effectuer des calculs d'itinéraire en voiture, en vélo, ou à pied à n'importe quel endroit couvert par le fichier d'entrée, c'est-à-dire sur n'importe quelle partie du territoire métropolitain.

Des limites pratiques s'appliquent. D'abord, la configuration du serveur OTP et les paramètres de recherche imposent une limite de durée de voyage, ici réglée à huit heures. Ensuite, les temps de calcul varient fortement en fonction de la distance entre le point d'origine et le point de destination, ainsi que le nombre de chemins alternatifs qui existent entre les deux points. Ainsi, une recherche à l'intérieur d'une ville principale a un temps de réponse de l'ordre de 100 ms, une recherche en petite couronne Parisienne prend environ 200 ms, mais des recherches d'une ville à l'autre peuvent prendre plusieurs secondes.

Ce n'est pas surprenant, car OpenTripPlanner n'emploie pas les techniques de pointe de recherche de plus court chemin de point à point sur la voirie. Il s'agit d'un choix conscient, car en tant que calculateur d'itinéraires multimodal fortement axé sur les transports en commun, OTP n'a pas comme vocation de produire de tels chemins vers un seul point à longue distance. Il existe de nombreux autres logiciels (dont plusieurs en open source tels que OSRM, GraphHopper, Valhalla) qui s'adressent à ce problème, qui a suscité un investissement massif bien avant les

techniques de calcul d'itinéraire TC. Le module de calcul d'itinéraire sur voirie de OTP est donc consacré quasi exclusivement au calcul d'une origine vers toutes les destinations atteignables, pour trouver simultanément des chemins vers tous les arrêts TC dans un rayon donné. Plus le rayon augmente, et moins on utilisera l'ensemble des chemins présents dans ce rayon de recherche, moins il est pertinent. Mais il sert à démontrer la présence et l'utilisation d'une représentation totale de la voirie nationale, qui servira à relier les arrêts TC entre eux et à fournir des itinéraires de rabattement vers les points d'accès TC, que ce soit à pied, en vélo, ou en VP en passant par des parcs relais.
La dernière étape pour finir la préparation du réseau voirie est de renommer le fichier `graph.obj` produit ci-dessus, lui donnant l'appellation `streetGraph.obj` pour indiquer à OTP qu'il s'agit d'une couche de base, à laquelle on ajoutera des données TC pour en faire le graphe final plus tard.



Quelques itinéraires en vélo (Nord Finistère, en haut) et à pied (à Marseille, en bas) qui démontrent le niveau de détail présent dans le graphe de la voirie.

Les données transports en commun

Nous avons commencé par trois fichiers choisis pour donner une couverture géographique permettant des recherches d'itinéraire sur de longues distances (Brest à Marseille en passant par Paris). Après avoir vu des résultats rassurants, nous avons rapidement élargi l'ensemble de données pour arriver à 34 fichiers d'entrée GTFS, choisis pour couvrir au moins les 100

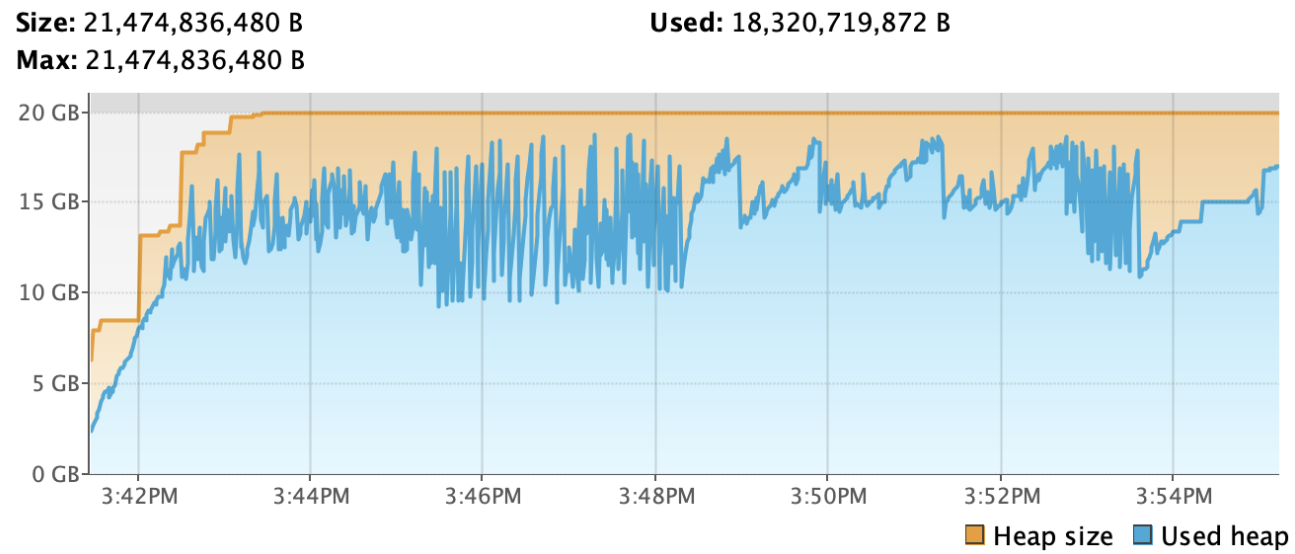
communes les plus peuplées de France. Nous avons privilégié les agrégats pré-fusionnés qui étaient disponibles pour toutes les grandes agglomérations sauf Aix-Marseille-Provence, dont les 15 flux GTFS séparés dominent la liste. Le téléchargement de ces flux, ainsi qu'un nettoyage minimal et la création d'un fichier de configuration OTP qui leur correspond est automatisé par des scripts présents dans le dépôt https://gitlab.com/CodeursEnLiberte/france-planner-xp/data. L'ensemble de ces fichiers sont disponibles au Point d'Accès National, d'où ils sont téléchargés par les scripts. Quelques jeux de données (notamment celui du Pays de la Loire) ont des problèmes de collisions d'identifiants. OTP tolère mal les identifiants doublons, donc ces erreurs sont réparées automatiquement par le script. Il faut préciser la localisation du graphe de la voirie lors du téléchargement pour l'inclure dans ce fichier de configuration OTP généré :

```
fxpdata --street-graph /chemin/du/fichier/streetGraph.obj
```

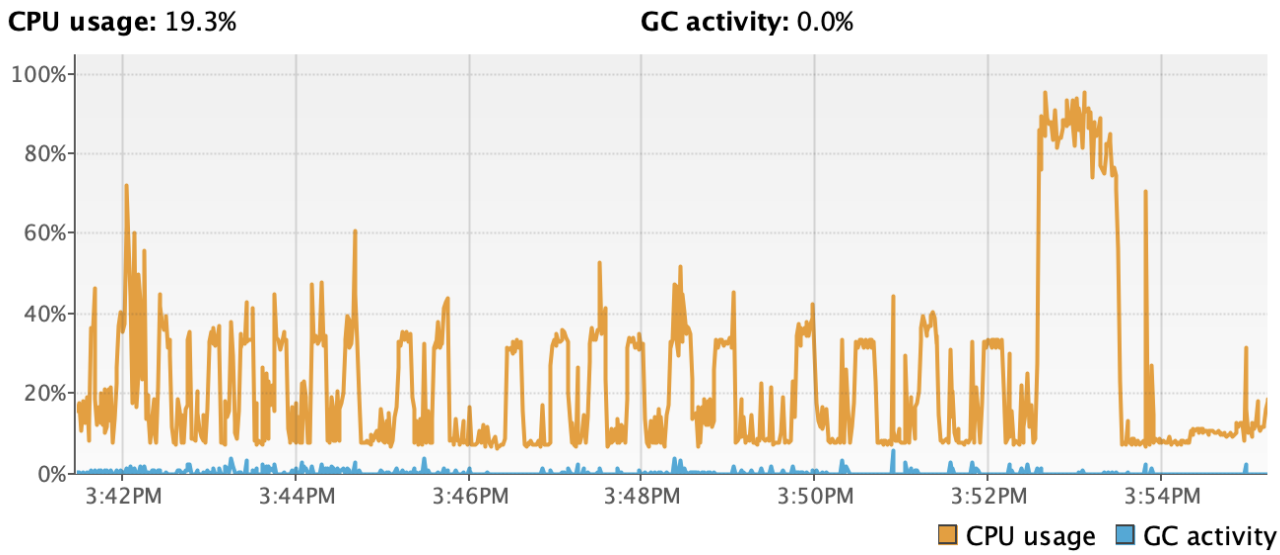
Ensuite nous procédons à la construction du graphe voirie plus TC :

```
java -Xmx30G -jar target/otp-2.5.0-SNAPSHOT-shaded.jar --loadStreet --save
../data/build/latest
```

L'option loadStreet dirige OTP à charger d'abord la couche voirie avant d'y ajouter les données TC, mais le chemin du fichier contenant le graphe de la voirie doit être présent dans le fichier de configuration créé par fxpdata. Avec un tas JVM de 25 Gio cette opération prend environ 18 minutes, et produit un fichier de sortie (`graph.obj`) de 3,6 Gio. Avec un tas réduit à 20 Gio la durée de l'opération reste essentiellement inchangée.



Consommation de mémoire vive lors de l'ajout des données TC au graphe voirie.



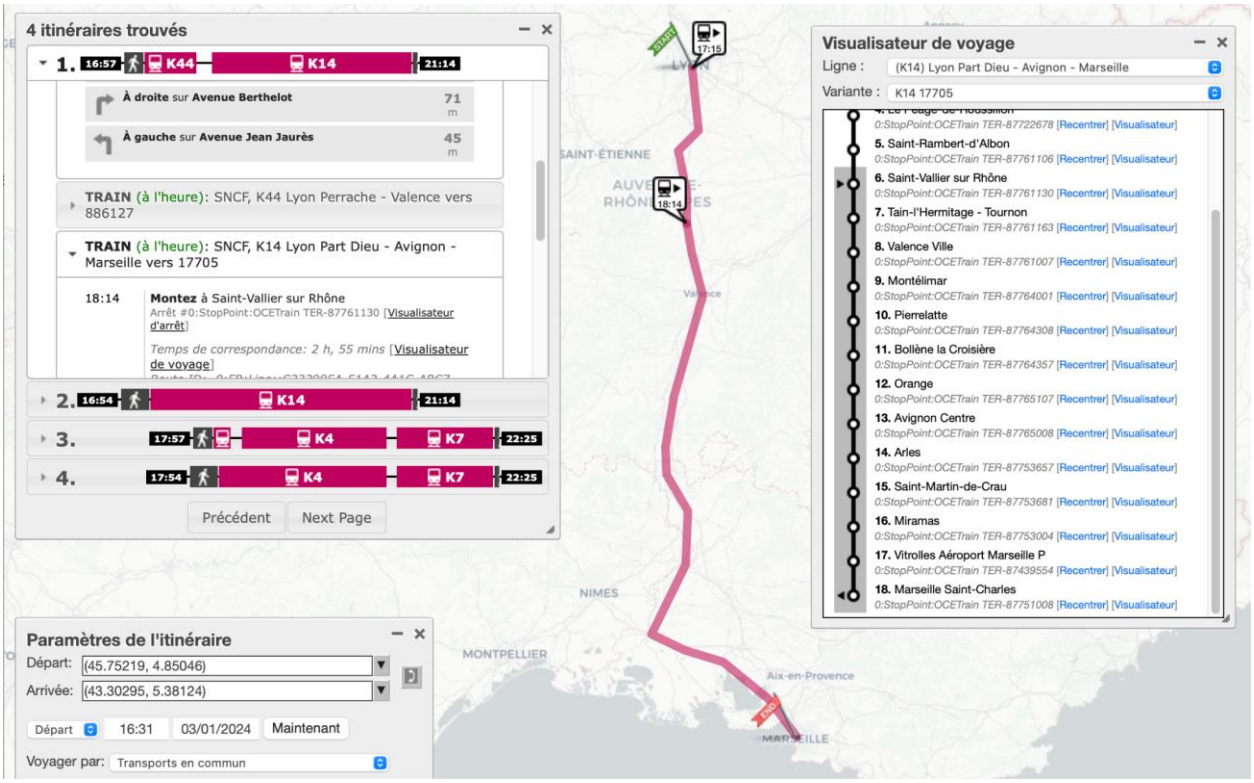
Activité de calcul et de ramassage (GC) lors de l'ajout des données TC au graphe voirie.

NB: l'ajout des couches TC doit se faire avec la même version de OTP qui a construit le graphe de la voirie. Par précaution contre des incompatibilités, les commits successifs de OTP sont habituellement rendus incapables de charger un fichier écrit par une version précédente. Toutefois, quand on est sûr qu'aucune incompatibilité n'existe au niveau du graphe de la voirie, il est possible de modifier la version acceptée par OTP, en modifiant l'élément otp.serialization.version.id dans le fichier otp-project-info.properties à la racine du JAR (par exemple, avec vi qui est capable de modifier des fichiers à l'intérieur d'un ZIP).

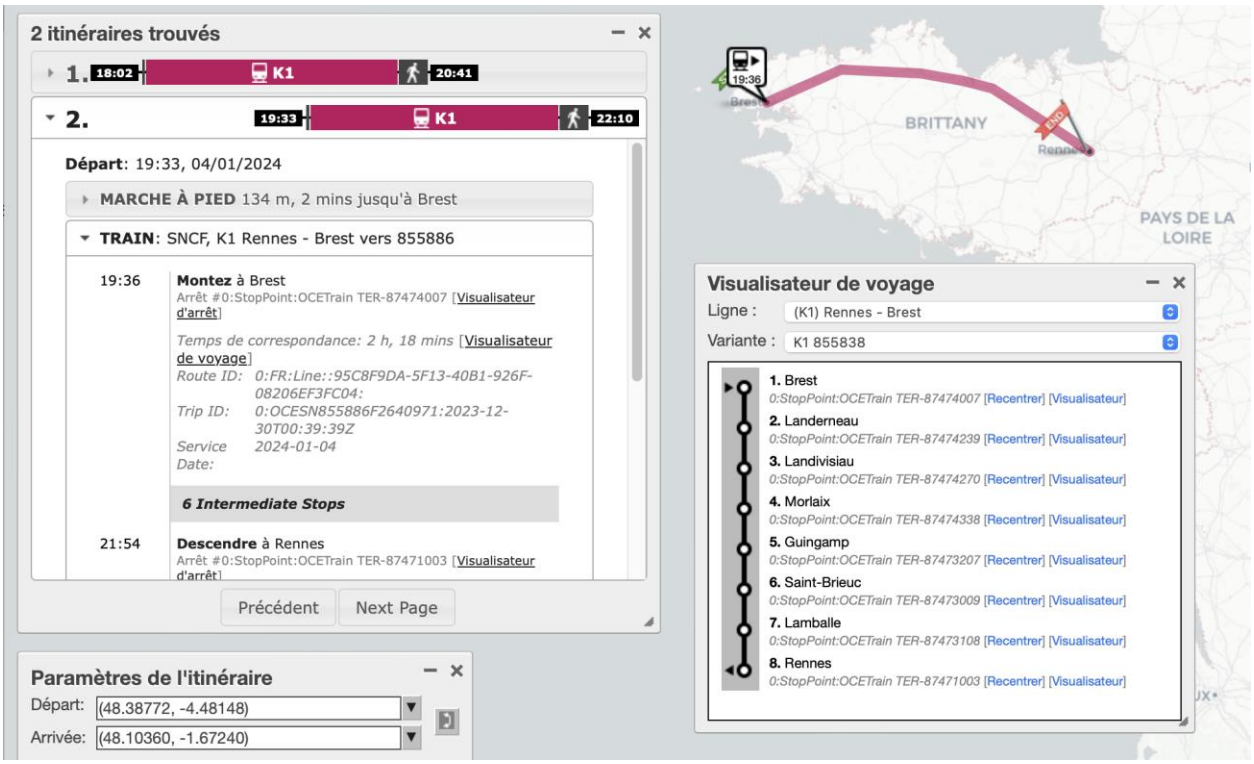
Une fois ce graphe voirie plus TC construit, on peut lancer le serveur OTP en indiquant le dossier qui contient le fichier graphe :

```
java -Xmx25G -jar target/otp-2.5.0-SNAPSHOT-shaded.jar --load --serve
../data/build/latest
```

Le démarrage prend environ deux minutes, dont une minute pour charger le fichier de graphe et une minute pour reconstruire des indexes de la voirie. Une fois le serveur actif, on peut effectuer des requêtes avec le frontal de débogage. Comme avec la voirie seule, on constate la possibilité de faire une recherche d'itinéraire dans n'importe quelle région sur un serveur unique.



Itinéraire de Lyon à Marseille, avec indications temps réel et rabattement à pied.



Itinéraire de Brest à Rennes en TER (même graphe et serveur que image ci-dessus).

La réponse peut contenir des dizaines d'itinéraires qui déborderont cette ancienne interface web. Il est donc conseillé d'introduire le paramètre 'numItineraries=5', soit dans l'élément de l'interface web prévu à cet effet, soit dans la configuration `router-config.json` du serveur lui-même.
Nous avons souvent testé cette combinaison voirie plus TC en appliquant les fichiers de configuration `router-config.json` et `otp-config.json` dans le dossier 'conf' du dépôt 'perf', qui sont dérivés de ceux utilisés en Finlande. La qualité et la rapidité des réponses obtenues avec les paramètres par défaut (sans ces fichiers de configuration) sont déjà tolérables.

Données temps réel

Le fichier de configuration `router-config.json` dans le dépôt `perf` contient une section `updaters`. Les modules qui y sont configurés cherchent périodiquement des données temps réel et les appliquent aux données TC statiques. Nous avons surtout testé les flux temps réel de la SNCF, qui consistent en trois flux GTFS-RT *TripUpdates* (prévisions des horaires de passage aux gares) et un flux SIRI-SX qui fournit des alertes textuelles à afficher aux voyageurs dont l'itinéraire comprend un véhicule spécifique.

Les flux GTFS-RT de la SNCF sont structurés différemment que son flux SIRI. Chacun des trois flux GTFS-RT correspond à un flux GTFS statique (TER, TGV, Intercités). Par contre, il n'y a qu'un seul flux SIRI-SX pour ces trois flux statiques. Ce flux SIRI-SX comprend donc des messages qui font référence à des entités telles que des trajets de véhicule qui appartiennent à trois ensembles indépendants. Or, OTP associe un *feedId* unique à chaque flux d'entrée et les *updaters* sont conçus pour associer les messages d'un flux temps réel aux entités d'un seul *feedId*.

Les messages SIRI-SX SNCF sont structurés différemment que ceux utilisés dans les pays nordiques. Notamment, l'identifiant de l'entité affectée par l'alerte se trouve à *Situation -> Affects* dans les données nordiques, alors qu'il se trouve à *Situation -> Consequences -> Affects* dans les données SNCF. Au lieu de créer des versions différentes des modules OTP capables de consommer des données nordiques ou français, l'idéal serait d'aligner les profils et les conventions d'API au niveau européen afin de concentrer l'ensemble de l'effort de développement logiciel sur des modules et des configurations standards.

Le flux SIRI-SX est fourni en 'SIRI Lite', qui signifie (entre autres) une API de style REST au lieu de SOAP. Le module SIRI-SX d'OTP est conçu pour faire des requêtes SIRI avec la méthode HTTP POST (en envoyant une *ServiceRequest* dans le corps de la requête), au lieu de la simple méthode HTTP GET attendu par le serveur proxy.transport.data.gouv.fr. Ce dernier répond au message POST de OTP avec une erreur 405 *Method not allowed.* OTP devra être modifié pour envoyer des GET et mieux coopérer avec des serveurs SIRI Lite. Nous avons déjà effectué cette modification qui pourra être proposée à la communauté OTP. Mais ce changement n'est pas suffisant en soi pour consommer ce flux SIRI-SX : il y a les problèmes plus épineux des espaces de noms des identifiants et de la structure des messages eux-mêmes.

Nous avons aussi examiné les données temps réel en Ile-de-France, décrites à https://transport.data.gouv.fr/datasets/reseau-urbain-et-interurbain-dile-de-france-mobilites ainsi que sur la Plateforme Régionale d'Information pour la Mobilité (PRIM), dans son catalogue des API à https://prim.iledefrance-mobilites.fr/catalogue-data et son PDF de documentation fonctionnelle à https://prim.iledefrance-mobilites.fr/content/files/2022/10/Documentation-fonctionnelle---Octobre-2022-2.pdf. IDFM propose des flux SIRI qui comportent les prévisions des horaires de passage. Toutes ces sources de documentation semblent décrire des données organisées par gare (prochains passages à chaque gare prise individuellement, SIRI-SM *Stop Monitoring*), alors que OTP est fait pour recevoir des données structurées par véhicule (un horaire de passage par gare visitée par chaque véhicule, SIRI-ET *Estimated Timetable*). Or, ces données SIRI-ET semblent exister, sans être mises en avant par la PRIM. En se basant sur le code source d'un logiciel FOSS existant (https://github.com/Tristramg/idfm_proxy) nous avons réussi à les demander avec :

```
wget --header "apikey: API_KEY" "https://prim.iledefrance-
mobilites.fr/marketplace/estimated-timetable?LineRef=ALL"
```

La réponse comporte 68 Mio de JSON avec pour chaque *Line* et *DatedVehicleJourneyRef* les *EstimatedCalls* comme par exemple :

```
"StopPointRef": { "value": "STIF:StopPoint:Q:29631:" },
"DepartureStatus": "ON_TIME",
"AimedDepartureTime": "2024-03-13T06:55:00.000Z",
"ExpectedDepartureTime": "2024-03-13T06:55:43.000Z",
"ArrivalStatus": "ON_TIME",
"AimedArrivalTime": "2024-03-13T06:55:00.000Z",
"ExpectedArrivalTime": "2024-03-13T06:55:43.000Z"
```

Comme pour les données SIRI de la SNCF, ce serveur s'attend à des requêtes SIRI Lite, mais OTP émet des requêtes SIRI classiques et s'attend à une réponse XML. Les modifications nécessaires à OTP sont relativement légères, mais ne sont pas encore faites donc nous n'avons pas encore testé l'intégration de ces messages SIRI-ET.

Mise en relation des identifiants temps réel et statiques

Les trois modules OTP configurés pour gérer les *TripUpdates* GTFS-RT de la SNCF arrivent à associer presque tous les messages reçus à une entité tirée des données statiques.

Intercités : 16 sur 18 messages appliqués aux données statiques. Les deux messages qui ont échoué avaient comme *ScheduleRelationship* ADDED et ont produit l'erreur TOO_FEW_STOPS.

TGV : 94 sur 96 messages appliqués. Les *ScheduleRelationship* et types d'erreur des deux messages échoués étaient :
CANCELED, NO_TRIP_FOR_CANCELLATION_FOUND ;
ADDED, TOO_FEW_STOPS.

TER : 1004 sur 1014 messages appliqués. Sept messages avec *ScheduleRelationship* ADDED rencontraient une erreur TOO_FEW_STOPS ; Trois messages avec *ScheduleRelationship* SCHEDULED rencontraient une erreur INVALID_STOP_SEQUENCE.

Pour traiter les alertes SIRI-SX nous avons utilisé une version d'OTP modifiée qui cherche à appliquer chaque *Situation* aux trois flux GTFS SNCF l'un après l'autre. Ainsi, sur les 530 alertes reçues, toutes sauf 60 trouvaient une entité statique pertinente.

Ensuite, nous avons testé une autre stratégie pour accommoder ce flux SIRI-SX : la fusion des trois flux statiques sous un seul identifiant *FeedId*. Cela ne passe pas forcément par la création d'un fichier d'entrée unifié ; Nous avons dirigé OTP à accorder le même *FeedId* (SNCF) aux trois fichiers d'entrée. Sur un serveur de test, les taux de réussite ainsi obtenus sont comparables au cas avec trois *feedId* différents, avec des modifications au code source d'OTP beaucoup plus simples.

Tant que tous les identifiants présents sont tirés d'un même espace de noms, il n'y aura pas de collision. Les identifiants des lignes et des trajets de véhicule dans les trois flux SNCF sont structurés de la même manière et semblent être uniques. Il est fort possible que ces trois fichiers sont essentiellement des fragments d'une seule base et partagent un espace de noms. Il faudrait vérifier auprès de la SNCF quelles garanties existent sur l'unicité de ces identifiants, pour confirmer si la fusion de ces trois jeux de données est effectivement triviale.

Dérive d'identifiants

Après un certain temps, OTP n'arrive plus à appliquer les messages temps réel reçus. Plus le temps passe, plus le taux de réussite baisse. Après une demi-journée, la moitié des messages ne trouvent plus d'entité statique à laquelle s'attacher.

Des discussions d'utilisateurs sur le Point d'Accès National indiquent qu'il y avait des problèmes d'identifiants temps réel pendant le mois de janvier 2024, au moment où nous travaillions sur ce prototype.
https://transport.data.gouv.fr/datasets/horaires-des-lignes-ter-sncf#dataset-discussions
Les problèmes que nous avons rencontrés étaient sans doute exacerbés par cette panne, mais persistent bien après la date de fin de l'incident décrit.

Le problème ne semble pas être dû à une dégradation progressive de l'état interne du serveur OTP. L'incapacité de mettre en relation messages temps réel et entités statiques se manifeste indifféremment, que ce soit un serveur qui reste allumé depuis un jour entier, ou bien un serveur qui vient d'être allumé avec un jeu de données statiques édité le jour précédant.

Notre hypothèse est que les identifiants d'une partie des trajets de véhicules sont modifiés d'une édition à l'autre du fichier GTFS statique, et que les identifiants dans les messages temps réel ne correspondent qu'aux identifiants employés par la version la plus récente des données statiques. Au premier regard, les identifiants semblent être conçus pour être stables. Cependant, les identifiants des *trips*, les trajets de véhicules, se terminent par une date et une heure au format ISO. On peut imaginer que c'est le résultat d'un *TimeDemandGroup* Transmodel qui a été instancié à une heure de départ précise pour s'exprimer en GTFS. Mais en regardant de plus près, on remarque que les horodatages ISO suffixés aux identifiants sont tous environ 35 minutes après minuit sur des jours différents. Il s'agit peut-être d'un identifiant de version mise à jour quand un trip est modifié dans la base d'origine. Il est quand même surprenant de voir la moitié des identifiants ainsi versionnés devenir obsolètes en l'espace de douze heures.
Examinons un exemple de plus près. Nous avons démarré un serveur hier, avec un jeu de données d'entrée qui date d'hier. Aujourd'hui, il reçoit un message GTFS-RT au sujet d'un *trip* TER qu'il ne reconnaît pas, avec comme identifiant `OCESN854826F2638230:2024-02-17T00:32:45Z`. Cet identifiant est facilement trouvable dans le GTFS statique d'aujourd'hui, mais n'existe pas dans l'édition d'hier. Notons que l'horodatage ISO est 32 minutes après minuit aujourd'hui même, entre la publication de l'édition d'hier et l'édition d'aujourd'hui. Si nous ne cherchons que le préfixe de l'identifiant (avant le premier deux points), il y a exactement un identifiant dans l'édition d'hier qui y correspond, avec un suffixe horodatage 24 heures plus tôt: `OCESN854826F2638230:2024-02-16T00:32:35Z`.

Ces *trips* dans les deux éditions sont presque identiques, sauf le *serviceId* et le *blockId.* L'identifiant du service a changé, mais signifie exactement les mêmes 11 jours sur les trois prochains mois. Les *stopTimes* qui y sont associés semblent être identiques. Dans les deux cas, c'est le seul trajet qui emploie son *blockId*. Pour résumer, au moins dans le cas des *trips,* les identifiants semblent intégrer un horodatage qui est souvent mis à jour même quand les détails (observables) du trajet n'ont pas changé.

La structure de ces identifiants est en partie expliquée dans une discussion sur le PAN:
https://www.data.gouv.fr/en/datasets/horaires-des-lignes-intercites-sncf/#/discussions
Mais ce n'est que la première partie de l'identifiant comprenant le numéro du train qui est expliqué, pas le bloc de chiffres et l'horodatage après la lettre F ou R.

Sur https://transport.data.gouv.fr/datasets/horaires-des-lignes-ter-sncf on apprend que pour le SIRI SX Lite "l'identifiant commun des circulations entre les différents flux est le numéro commercial du train, que vous trouverez dans <VehiculeJourneyRef>". Celui-ci se trouve dans la *AffectsScopeStructure* examinée par OTP pour associer le message aux entités statiques, donc a priori nous utilisons le bon identifiant.

L'invalidation progressive des identifiants posera problème. S'il est possible et souhaitable de reconstruire le graphe à chaque fois qu'une nouvelle édition des données GTFS est publiée, il sera quasi impossible de basculer vers ce nouveau graphe à précisément le moment où les données temps réel commencent à employer les nouveaux identifiants. Soit il faudra accepter

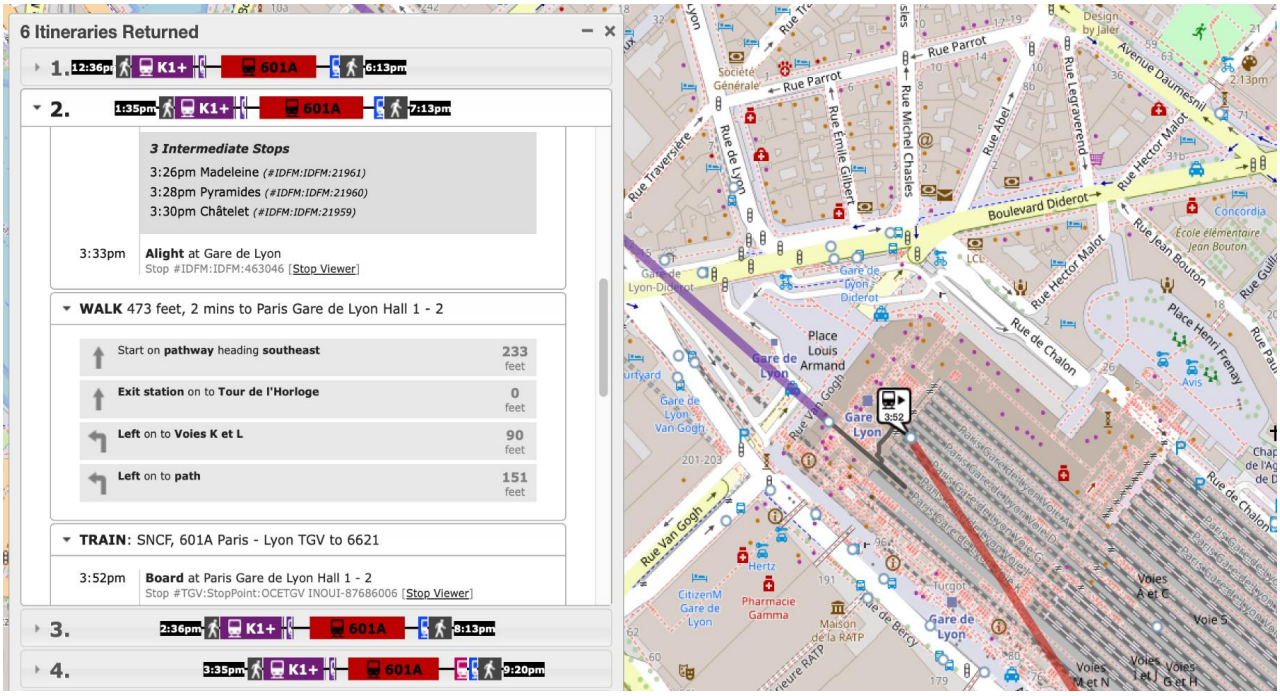
une panne effective du système temps réel pendant un certain temps à chaque fois qu'une nouvelle édition des données statiques est éditée, soit OTP devra être modifié pour extraire et comparer des sous-chaînes ou des préfixes d'identifiants pour rendre les recherches d'entités plus souples, et ainsi mieux faire le pont entre deux jeux de données statiques successifs.

Les *updaters* OTP ont une fonctionnalité *fuzzyTripMatching* qui est, au premier regard, prometteuse, mais elle n'est pas faite pour détecter les préfixes ni d'autres sous-chaînes des identifiants.

Correspondances

Correspondances verticales surface-métro

OTP superpose et relie les sources de données entre eux de manière heuristique. En surface, la voirie OSM fournit des chemins réalistes. Mais en l'absence d'une étape supplémentaire d'intégration de données qui intègre une micro-cartographie des gares décomposée verticalement en niveaux, les correspondances entre jeux de données indépendants peuvent poser problème. Le cas le plus courant est une correspondance entre une station de métro en profondeur, tirée d'un jeu de données intégré mais local, et une gare grandes lignes tirée d'un jeu de données à l'échelle nationale. À» titre d'exemple, voici un itinéraire qui comprend une correspondance entre la ligne 14 du métro parisien et un TGV SNCF. Le temps de correspondance de 2 minutes est clairement irréaliste, mais OTP n'a pas suffisamment d'informations pour comprendre qu'il y a un dénivelé important (et beaucoup de tunnels) entre ces deux points.



La solution heuristique est de spécifier le paramètre subwayAccessTime (https://docs.opentripplanner.org/en/v2.3.0/BuildConfiguration/#subwayAccessTime). Les scripts dans le dépôt *data* ajoutent ce paramètre automatiquement au `build-config.json` avec une valeur de 5 minutes. Néanmoins, on trouve facilement des itinéraires comprenant un rabattement de deux ou trois minutes à pied avant un trajet en métro, où ce délai de 5 minutes n'est pas pris en compte. La fonction `StreetTransitStopLink#getStreetToStopTime` ignore le *subwayAccessTime* si un arrêt possède des *pathways*, des éléments facultatifs GTFS décrivant les cheminements nécessaires pour accéder à un quai à partir d'une bouche de métro. Or, le flux GTFS de Ile-de-France Mobilités possède un tableau *pathways*. Il est possible que les tunnels d'accès de ces stations de métro y soient décrits, mais avec des longueurs ou des temps de traversée insuffisants.

Modèle numérique de terrain (MNT)

OTP est capable d'appliquer un MNT au moment de la construction du graphe, pour ensuite prendre en compte l'effort relatif des segments vélo et marche à pied pendant la recherche d'itinéraires. Nous avons identifié le jeu de données RGE ALTI de l'IGN qui semble être pertinent, mais nous n'avons pas tenté de l'intégrer au graphe. OTP est capable de gérer des MNT comme celui-ci qui sont découpés en tuiles, mais à cette échelle et à cette résolution les données sont très volumineuses. Les utilisateurs OTP en Norvège et en Finlande appliquent des MNT et pourraient fournir des conseils à ce sujet.

- https://geoservices.ign.fr/rgealti
- https://medium.com/@gui.attard/pre-processing-the-dem-of-france-rge-alti-5m-for-implementation-into-earth-engine-de9a0778e0d9

Interfaces Graphiques

OpenTripPlanner. Entretien avec Leonard Ehrenfried (développeur OpenTripPlanner qui gère plusieurs déploiements OTP en Allemagne, <mail@leonard.io>) complété d'observations personnelles et d'expériences avec les logiciels mentionnés. Ces conclusions ont été contribuées à la documentation officielle du projet OpenTripPlanner, pour les rendre plus facilement disponibles.

OpenTripPlanner est un système client-serveur. Un service backend écrit en Java est accessible via une API par un logiciel client, qui est de façon générale intégré dans une page web ou une application mobile. Lorsque les développeurs OpenTripPlanner font référence à OpenTripPlanner ou OTP, ils parlent souvent spécifiquement du service backend Java. Mais ce service n'est utile aux utilisateurs grand public qui souhaitent planifier leurs voyages qu'à travers une interface web reliée à quelques autres composants cartographiques et de recherche d'adresses.

Dans le cadre de ce document et des discussions sur le développement de OpenTripPlanner, les termes front, UI et client seront utilisés de manière équivalente. En réalité, de nombreux types de

clients sont possibles, qui ne sont pas tous des interfaces utilisateur. L'API OpenTripPlanner peut être appelée par divers scripts et microservices, potentiellement au sein de ce que l'on appellerait traditionnellement un "backend" sans aucune interface utilisateur. Par exemple, les mises à jour des plans de voyage pourraient être demandées par un client, résumées et envoyées sous forme de messages texte sur un réseau mobile. Mais ici, nous parlons exclusivement d'interfaces utilisateur graphiques, généralement écrites en Javascript pour être exécutées et affichées dans les navigateurs web.

Il existe deux types d'interface OTP : les frontaux de débogage et les frontaux de production.

D'une part, les **frontaux de débogage** sont inclus dans le dépôt principal de OpenTripPlanner, et sont conçus pour fonctionner sans paramétrage particulier. Ils sont servis par OTP lui-même ou par un simple serveur web local, ou sont configurés de manière à rendre un déploiement unique utilisable par tous via un réseau de diffusion de contenu (CDN). L'objectif principal des interfaces de débogage est de permettre aux développeurs OTP (ainsi qu'aux personnes qui évaluent OpenTripPlanner pour leur organisation) d'observer, de tester et de déboguer l'instance OTP sur laquelle ils travaillent. Les frontaux de débogage exposeront certains détails internes du fonctionnement d'OTP, tels que les nœuds et les arêtes du graphe, les permissions de traversée, ou les identifiants des données TC d'entrée. Leur rôle principal est de servir d'outil de développement et de déploiement.

D'autre part, les **frontaux de production** sont un composant des déploiements OTP plus importants, tournés vers le grand public, avec une apparence et une expérience utilisateur plus soignées. Ils ne sont pas "prêts à l'emploi" et en général nécessitent une grande quantité de configuration et de coordination avec des composants externes tels que les serveurs de tuiles cartographiques et les géocodeurs. Ils sont conçus comme des composants d'un déploiement OTP entretenu et géré par des professionnels, et présentent une vue plus simple des options et des résultats d'OpenTripPlanner à un public d'utilisateurs finaux non techniques.

Les frontaux de débogage

Le dépôt principal d'OpenTripPlanner contient actuellement deux interfaces web de débogage : l'originale à https://github.com/opentripplanner/OpenTripPlanner/tree/dev-2.x/src/client et une plus récente en cours de développement à https://github.com/opentripplanner/OpenTripPlanner/tree/dev-2.x/client-next.

Le **client de débogage original** est une interface utilisateur basée sur jQuery et Backbone dont l'histoire remonte à plus d'une décennie, aux premiers jours du projet OTP. Il se connecte au backend Java OTP via une API REST utilisant le vocabulaire GTFS. Historiquement, elle servait de l'interface par défaut d'OTP, et elle continue d'être disponible sur toute instance d'OTP à l'URL racine.

Le **nouveau client de débogage** est une application à page unique React/TypeScript (SPA) qui peut être servie localement ou via un réseau de diffusion de contenu (CDN). Contrairement au

client de débogage original, il se connecte au backend Java via une API GraphQL plus récente en utilisant le vocabulaire Transmodel (celui utilisé par NeTEx). Il est actuellement en cours de développement, mais devrait remplacer le client de débogage original une fois qu'il aura atteint une fonctionnalité équivalente.

Ces clients peuvent profiter du géocodeur de test qui est activé dans `otp-config.json` de la manière suivante :

```
// otp-config.json
{
  "otpFeatures" : {
    "DebugClient" : true,
    "SandboxAPIGeocoder" : true
  }
}
```

Les frontaux de production

Il existe de nombreuses interfaces de production différentes. Un certain nombre d'autorités de transports et de sociétés de conseil peuvent avoir créé de nouveaux frontends, que ce soit en Javascript ou sous forme d'applications mobiles natives, sans que l'équipe de développement d'OpenTripPlanner n'en ait connaissance. Ceci dit, il existe deux interfaces utilisateur principales qui sont recommandées par les membres de l'équipe de développement OTP qui travaillent sur des déploiements professionnels et publics d'OpenTripPlanner :
- Digitransit UI (https://github.com/HSLdevcom/digitransit-ui), qui fait partie du projet finlandais Digitransit.
- OpenTripPlanner React UI(https://github.com/opentripplanner/otp-react-redux), développée et maintenue par Arcadis IBI (https://www.ibigroup.com) transit routing team (https://www.ibigroup.com/ibi-products/transit-routing/).

Digitransit est un projet public open-source, créé en Finlande pour remplacer les solutions existantes de planification d'itinéraires à l'échelle nationale et régionale en 2014. Digitransit a été utilisé dans le monde entier, par exemple en Allemagne. Il s'agit d'un projet commun de l'autorité de transport d'Helsinki (HSL), de Fintraffic et de Waltti Solutions.

Pendant plusieurs années, **Arcadis IBI** a développé, déployé et maintenu des instances de l'interface utilisateur OTP basée sur React, contribuant systématiquement leurs améliorations aux dépôts originaux sous l'organisation OpenTripPlanner :
- https://github.com/opentripplanner/otp-ui
- https://github.com/opentripplanner/otp-react-redux

Le déploiement d'un système entier basé sur OpenTripPlanner, y compris une interface web, peut s'avérer assez complexe. Étant donné que le déploiement et la gestion d'un tel système exigent un engagement à long terme de la part d'une équipe professionnelle disposant d'une expérience

et d'une connaissance suffisantes des éléments constitutifs, les principaux projets de front-end de production n'ont pas vraiment pour objectif d'être "prêt à l'emploi". Malheureusement, cela constitue un obstacle pour les organisations qui cherchent à évaluer ces composants. Bien qu'on fasse actuellement des efforts pour rendre cette configuration plus simple, en particulier dans le cas des projets IBI Arcadis, il est toujours fortement recommandé de collaborer avec un spécialiste. Le processus prend au moins plusieurs jours pour un professionnel expérimenté, et dans le cas de Digitransit-UI peut rapidement devenir interminable pour celui qui n'est pas conscient des pièges. Ces deux systèmes s'appuient sur de nombreux services complémentaires en dehors du backend Java OTP. On doit configurer au moins une source de tuiles cartographiques et un géocodeur, mais il est courant d'avoir des dizaines de microservices complémentaires. Ce n'est pas obligatoirement aux auteurs des logiciels qu'il faut faire appel - certains développeurs OTP et petits bureaux de conseil ont toutes les compétences nécessaires.

Évaluation d'OTP-React-Redux

Nous avons réussi à rendre l'interface OTP-React-Redux utilisable en apportant quelques modifications au fichier de configuration exemple. Pour un déploiement de test hors production, il est possible d'utiliser les jeux de tuiles préconfigurés dans la section `map.baseLayers` du fichier `example-config.yml`, ainsi qu'un compte de test fourni par l'équipe du géocodeur Pelias à l'adresse https://geocode.earth. Ce compte de géocodage temporaire d'une durée d'environ deux semaines est suffisant pour tester le système à court terme. L'URL de leur serveur Pelias et la clé API temporaire sont substitués dans le fichier `example-config.yml` dans les sections `geocoder.apiKey` et `geocoder.baseUrl`. On doit également mettre à jour le périmètre et le point focal du géocodeur dans la même section.

Les éléments 'host' et 'port' au début de ce fichier de configuration doivent être configurés, et il est essentiel de décommenter l'élément 'v2 : true', en s'assurant qu'il reste indenté sous la section 'api'. OTP-React-Redux ne supporte plus l'API v1 et échoue avec une erreur de console Javascript si cet élément v2 n'est pas activé.

La configuration exemple n'active ni le mode métro, ni le mode ferry. Il suffit d'ajouter les lignes suivantes dans la section `modes:transitModes:`

```
    - mode: SUBWAY
      label: Subway
    - mode: FERRY
      label: Ferry
```

Ces chaînes de caractères seront traduites dans l'interface quand une langue autre que l'anglais est sélectionnée. L'internationalisation du texte est activée en décommentant la section 'language' et les langues désirées qui y sont imbriquées. Il suffit de décommenter le nom de la section, l'abréviation clé, et le nom d'affichage ainsi :

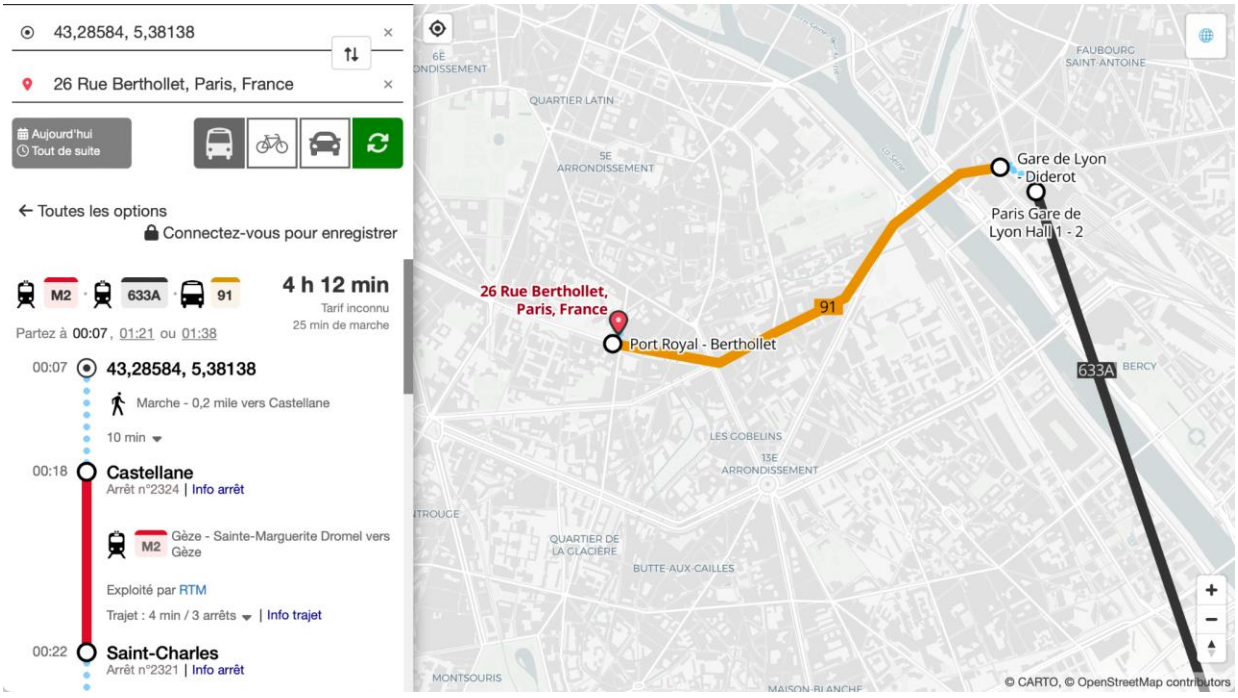
```
language:
```

```
fr:
  name: Français
vi:
  name: Tiếng Việt
```

Une fois l'internationalisation activée, la langue peut être choisie à l'aide de l'icône terre en haut à droite de l'interface web. En interne, OTP-RR cherche un code de langue ISO (tel que `fr` ou `es`) d'abord sous la clé 'lang' dans window.localstorage, puis dans le variable navigator.language (configuré dans les préférences du navigateur lui-même), avant de se rabattre sur l'élément `localization:defaultLocale` défini dans `example-config.yml`.

Les distances seront affichées en unités dites impériales (pieds et milles). OTP lui-même emploie les unités internationales métriques dans ses réponses, qui sont convertis et formatés par le front. Bien que le code source OTP-RR comporte des fonctions capables d'émettre des descriptions dans les deux systèmes, la fonction `humanizeDistanceString` semble toujours être invoquée avec le paramètre `outputMetricUnits=false` codé en dur. Il faudrait prévoir quelques modifications et contributions au code source OTP-RR pour rendre ce choix configurable.

Une fois configuré et lancé (yarn install; yarn start), on peut accéder à l'interface utilisateur dans un navigateur:



Itinéraire bout-en-bout Marseille-Paris, visualisé dans l'interface OTP React-Redux en français.

La configuration étant validée, on peut compiler une version statique optimisée et la publier. Pour le serveur de démonstration, après l'étape de compilation optimisée (yarn build) nous avons

installé le serveur web nginx et déplacé le dossier de sortie à la racine du contenu web (rm /var/www/html; mv dist /var/www/html).

Les observations et les astuces décrits ci-dessus ont été contribué à la documentation du projet OTP-RR pour faciliter les essais futurs.

Contrairement à OTP-RR, nous n'avons pas réussi à déployer tous les composants nécessaires de l'interface Digitransit. Ce sera vraisemblablement plus complexe que OTP-RR, mais le produit final visible en Finlande est prometteur.

Tarification et Billetterie

Entretien avec Andreas Tryti (Entur, <andreas.tryti@entur.org>)

Entur place OTP au centre d'un système de billettique qui comprend quelque 80 microservices. Environ la moitié de ces services concernent l'ingestion et le nettoyage des données, et l'autre moitié assure la disponibilité des services de billettique et de calcul d'itinéraire, avec OTP au milieu des deux.

Les contraintes contractuelles et juridiques sur la production et la livraison des données d'entrée sont essentielles au bon fonctionnement de l'ensemble.

- Utilisation d'un registre national des arrêts, avec zonage qui correspond aux tarifs
- Tout fichier reçu doit être valable pendant les 120 prochains jours (minimum)

Il y a un grand effort de validation et de mise en relation des données. Certaines erreurs dans le NeTEx d'entrée, même autour des identifiants, peuvent avoir des effets graves sur les processus de tarification et la billettique.

L'équipe responsable de OTP n'est qu'une équipe de 10 personnes, parmi 18 autres équipes qui s'occupent de la billettique. Cette équipe recherche d'itinéraire est beaucoup plus focalisé sur l'automatisation que les autres ; en réalité avec un niveau constant d'automatisation entre les équipes ce déséquilibre serait moins fort.

Le système est hébergé chez Google Cloud, et l'hébergement est coûteux à plus de 100k EUR par mois, mais le système est aussi très utilisé. Ils viennent de passer le cap de 1 milliard de requêtes OTP en janvier 2024. Ce système est désigné infrastructure numérique critique nationale, ce qui impose un certain exigence de fiabilité, et un coût qui correspond aux neufs supplémentaires après la virgule. Ils ont aussi été les premiers à employer NeTEx et SIRI à cette échelle. Ce niveau d'ambition n'est pas forcément obligatoire, et les coûts (hébergement et développement) pourraient être inférieurs.

Entur utilise uniformément les formats NeTEx. D'un point de vue technique, ce n'est pas forcément nécessaire pour le côté calcul d'itinéraire, mais respecte les attentes au niveau européen, et NeTEx comprend un volet produits et tarification qui n'est pas présent dans le GTFS.

Entur a une base de données "produits et tarifs" faite maison. Le processus d'entrée de données reste très manuel. Il devient progressivement plus automatique par le biais des données NeTEx, mais reste très manuel. Cette base comprend la configuration des trains, les catégories de sièges disponibles, la structure des prix (augmentation des prix en fonction de la demande) etc.

L'utilisateur effectue une recherche d'itinéraire OTP, le système demande les prix à associer aux options, puis les options différentes sont présentées dans l'appli. Quand une option est choisie, elle est ensuite envoyée aux services de réservation et d'achat.

De façon générale, Entur *ne fait pas appel* à des services (API) externes ni pour renseigner les tarifs, ni pour réserver des sièges. C'est important : en Norvège, avec la réforme des chemins de fer, la responsabilité pour la billettique a été officiellement inversée. Entur lui-même est l'autorité qui suit la disponibilité des sièges et les prix associés, et les chemins de fer et autres opérateurs sont obligés à utiliser les services de réservation d'Entur. Des exceptions existent pour les services internationaux ou à l'étranger (en Suède par exemple).

Les tarifs ne sont jamais estimés à l'intérieur de OTP, et ne sont pas pris en compte lors de l'optimisation et la génération des itinéraires. Les tarifs sont toujours calculés relatifs aux itinéraires déjà choisis, et y sont attachés en aval de OpenTripPlanner. Actuellement ce composant prend la forme d'un proxy qui attend l'ensemble d'une réponse OTP pour ensuite la "décorer" avec les informations tarifaires cherchés dans un autre service. Entur signale que cette architecture impose des latences inacceptables, et conseille pour toute nouvelle implémentation une architecture qui opère plutôt sur des flux de messages (itinéraires) individuels, qui arriveraient chez l'utilisateur final un par un.

Il n'y a aucun effort à l'intérieur de OTP pour garantir des tarifs variés. Par contre, la neutralité des itinéraires proposés est un élément important du calcul d'itinéraire, et tend à produire une grande quantité de réponses qui utilisent des modes et des opérateurs variés. Les paramètres de neutralité semblent suffisants pour garantir une variété de tarifs. Au pire, ces paramètres peuvent être ajustés pour émettre plus d'itinéraires longs mais moins chers. Même à long terme, ils ne voient pas le calcul des tarifs précis comme pertinent pour le composant OTP. Au plus, le calculateur d'itinéraires devra prendre en compte de manière abstraite les forces qui poussent les tarifs vers le haut ou vers le bas.

La latence et le temps de calcul ajoutés par ce processus de post-décoration d'itinéraires avec des tarifs ne posent pas de souci. Actuellement les requêtes sont parallélisées mais ne sont pas encore pleinement optimisées, et ne prennent que quelques centaines de millisecondes. Leur approche n'est pas d'aider l'utilisateur à trouver l'option le moins cher, mais de fournir toutes les options disponibles, et laisser l'utilisateur choisir (avec l'aide une appli) entre ces prix.

Les itinéraires trouvés par le calculateur (OTP) ne sont pas filtrés selon la disponibilité des sièges. Un train complet est quand même affiché, mais sans possibilité d'achat ou avec une indication qu'il ne reste que des places debout. Hypothétiquement, tous les itinéraires pourraient utiliser un même train complet en premier maillon, et il n'y a aucun mécanisme pour relancer la recherche en évitant ce train. Encore une fois, le mécanisme de neutralité joue un rôle. A terme il y a une volonté d'injecter des informations sur l'affluence (historique, prévisions, et temps réel) dans le processus d'optimisation d'itinéraires.

En dehors de OTP, aucun des composants de ce système n'est actuellement open source. En principe, les responsables sont favorables à l'ouverture du code source et à la collaboration avec d'autres organismes publics. Leur propriétaire (le ministère des transports) veut bien voir Entur promouvoir ses idées et ses méthodes à l'international. Mais en pratique jusqu'à présent, en dehors du calcul d'itinéraire une collaboration open source ne s'est pas produite. Les raisons sont multiples, et comprennent:

- Manque de familiarité avec le concept open source
- Manque de temps et de ressources face aux dates limites qui s'approchent
- Instinct d'éviter ce qui est inconnu ou imprévisible

Le modèle de déploiement et de conteneurisation d'Entur est déterminé plus par des exigences d'uniformité entre les 19 équipes, et pas par les caractéristiques d'OTP ou de la recherche d'itinéraires eux-mêmes. Leur approche n'a rien de particulier par rapport aux autres déploiements conteneurisés contemporains en centre de données. Ils ne voient pas un vrai avantage à collaborer sur cet aspect du système.

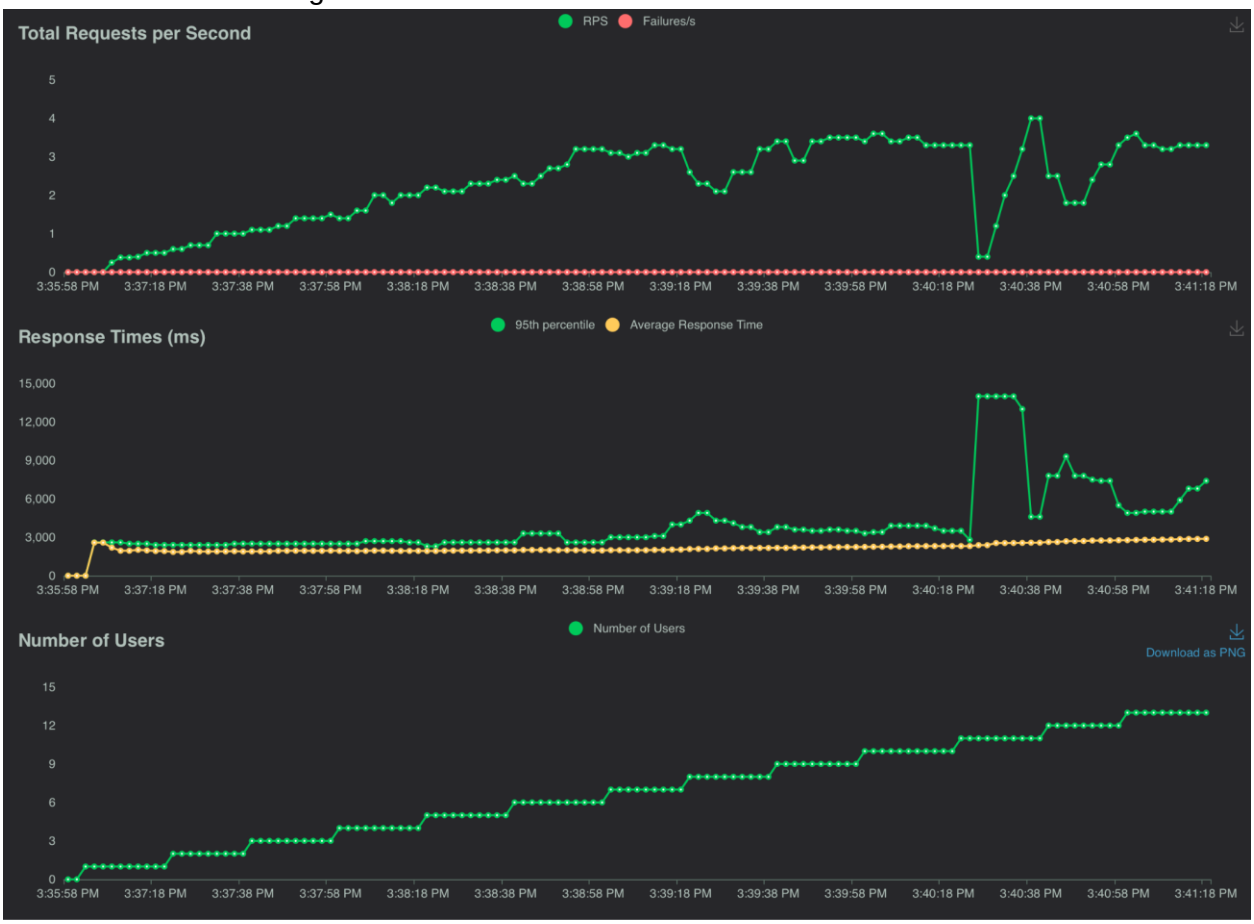
Pour conclure, Entur constate qu'une grande partie de la complexité de leur système est dû au besoin d'intégrer les anciens supports (billets, cartes d'abonnement). Actuellement leurs billets bout-à-bout sont essentiellement des paniers de billets liés aux anciens supports spécifiques aux opérateurs. Ils préféreraient avoir un système de comptes utilisateurs propre à Entur, pour consolider l'authentification du billet unifié chez eux (un seul code QR par exemple, ou un compte lié à la carte d'identité du voyageur). Cet objectif "account-based ticketing" est prioritaire et fait l'objet d'une collaboration avec les agences finlandais. *C'est ce composant du système qui est le plus susceptible d'être développé en open source.* Pour tout nouveau système de billettique national, Entur conseille fortement la création d'un système en rupture avec les supports et les systèmes de vente existants, et poursuit une coopération internationale sur ce sujet.

Tests de Performance

Les tests de performance décrits ci-dessous ont été menés avec un ensemble de paramètres OTP standardisés, dont beaucoup de paramètres par défaut. Le paramétrage du moteur de calcul peut altérer fortement les résultats (à la fois les itinéraires et les performances) ; nous reviendrons à cette question plus tard.

D'abord nous avons regardé (sur le serveur A décrit ci-dessous) un petit ensemble non-aléatoire de requêtes à l'intérieur de Paris et de Marseille. Nous avons augmenté la quantité de requêtes

simultanées d'un à douze, passant de 12% à 150% de la quantité de cœurs disponibles dans la machine de test. Les résultats correspondent aux attentes : à partir de 6 requêtes simultanées, le débit plafonne. À partir de huit requêtes simultanées, le débit baisse et le 95è centile du temps de réponse augmente. À partir de 11 requêtes simultanées, le serveur est en surcharge et le 95è centile est fortement dégradé.

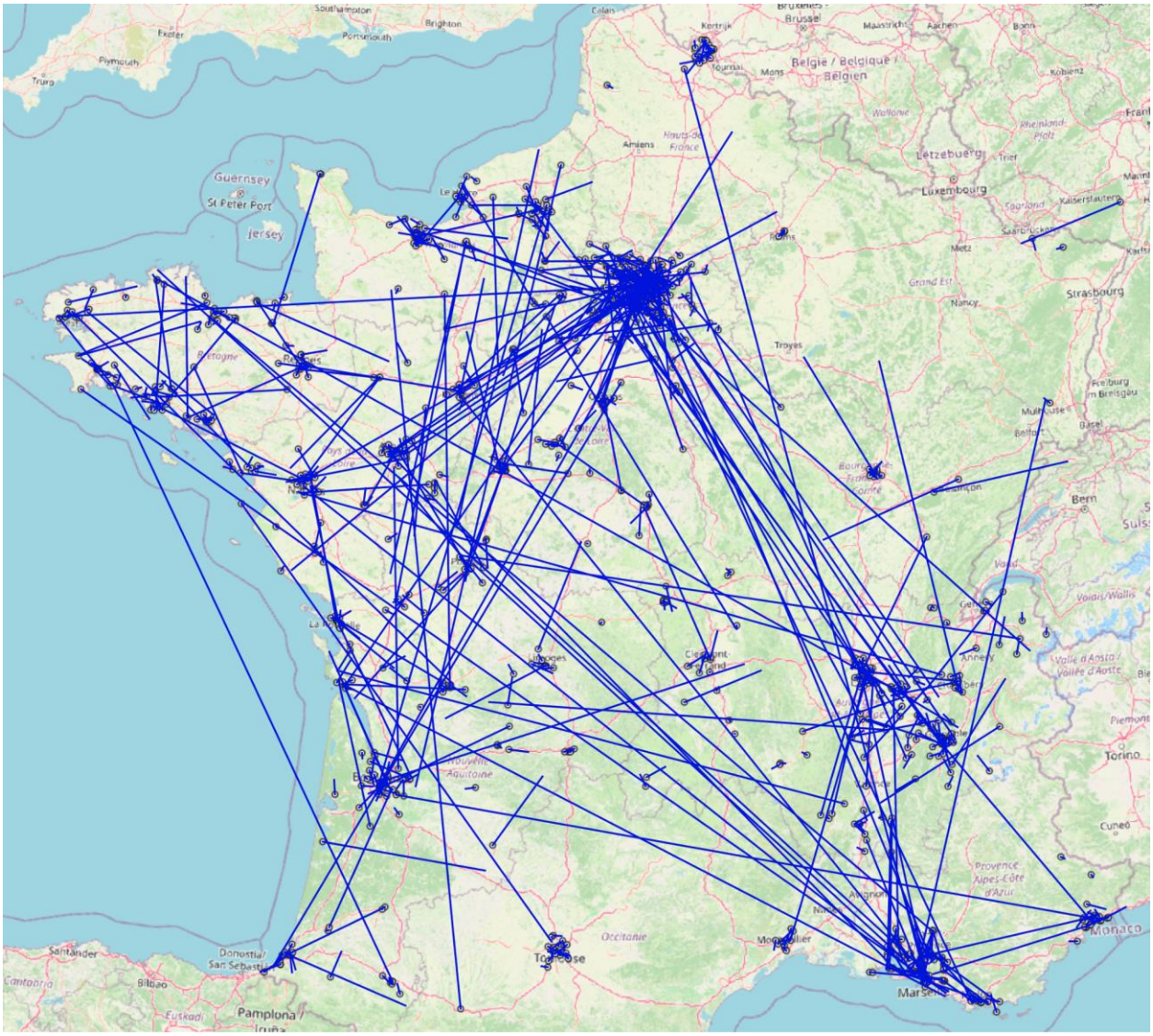


Montée en charge progressive, ratio utilisateurs-coeurs de 0 à 150%.

D'un certain point de vue, l'augmentation quasi-uniforme du débit jusqu'à sept clients simultanés est surprenant car il y a sans doute beaucoup de contention mémoire, et habituellement le bus mémoire est le maillon le plus lent dans les systèmes actuels. Mais ces résultats correspondent à ce que nous avons mesuré il y a quelques années sur des versions différentes de OTP et sur du matériel divers : une scalabilité quasi-linéaire, proportionnelle à la quantité de cœurs, tant que l'un des cœurs est laissé libre pour l'activité de ramassage (GC) de la JVM.

Demande synthétique

Ensuite nous avons voulu continuer les tests avec un ensemble de requêtes plus variées, à la fois en termes de localisation géographique, de distance, et des modes de transports empruntés. L'objectif est de construire des itinéraires qui touchent à tous les jeux de données GTFS individuellement et en combinaison, et qui touchent des zones du graphe distants les uns des autres en même temps pour réduire les effets de localité mémoire.

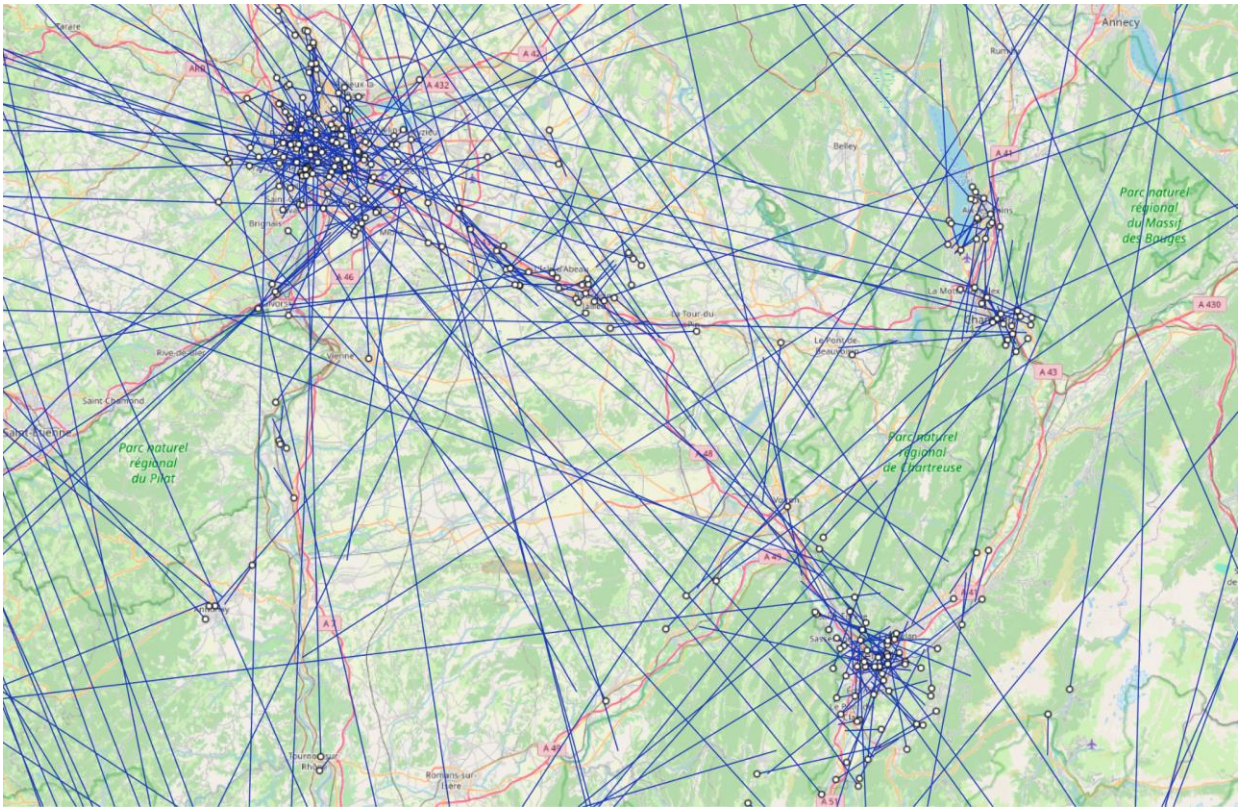


Trajets synthétiques à distribution de distance asymétrique.

Les points d'origine sont d'abord choisis parmi l'ensemble des arrêts TC présents dans le graphe (qui sont distribués plus ou moins proportionnellement à la densité de la population) et leurs coordonnées perturbées avec du bruit gaussien d'un écart-type de 300 mètres. Pour chaque point d'origine, une destination est choisie de la même façon, mais ces destinations sont retenues ou rejetées sélectivement pour assurer une distribution des rayons à vol d'oiseau asymétrique, biaisée vers les distances courtes, tout en respectant une distance minimale d'un kilomètre. Le calculateur ne trouve aucun itinéraire (même en marche à pied) pour environ 10% de ces requêtes, et si nous ne retenons que les requêtes qui produisent au moins un itinéraire en transports en commun, nous en perdons au moins 30%.

Pour évaluer les temps de réponse et la charge potentielle des serveurs dans une situation hypothétique où la grande majorité des requêtes activent le module TC de OpenTripPlanner, nous avons filtré ces requêtes aléatoires pour ne retenir que les 68% qui aboutissent à au moins un itinéraire TC. Un ensemble de 5000 paires origine-destination aléatoires sont ainsi réduit à

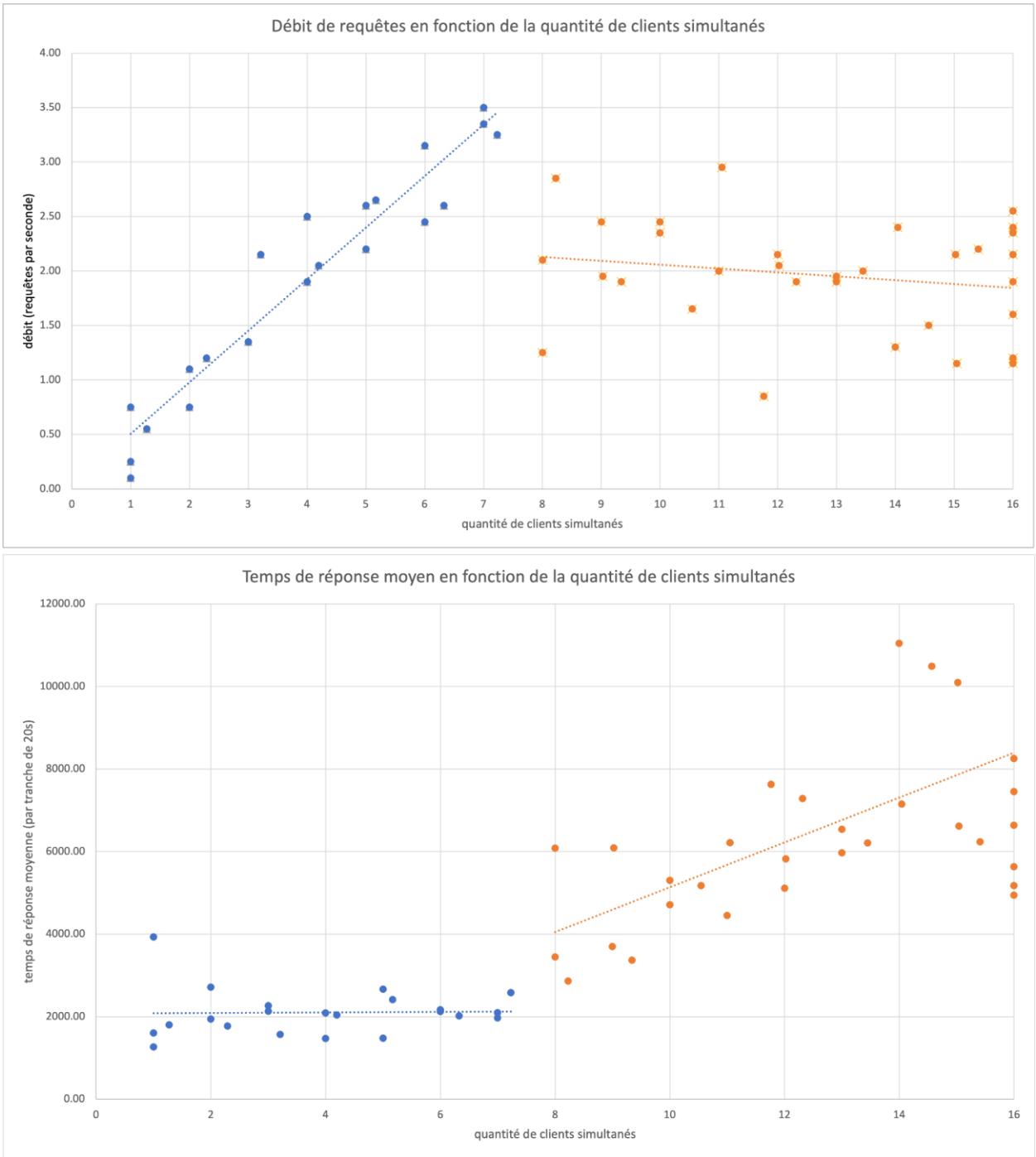
environ 3400 paires, qui sont utilisées en boucle de façon déterministe lors des tests de performance. Cf. les dépôts cités ci-dessus pour les données de sortie et les résultats des tests.



Paires origine-destination synthétiques retenues autour de Lyon, Grenoble, et Chambéry.

Serveur A

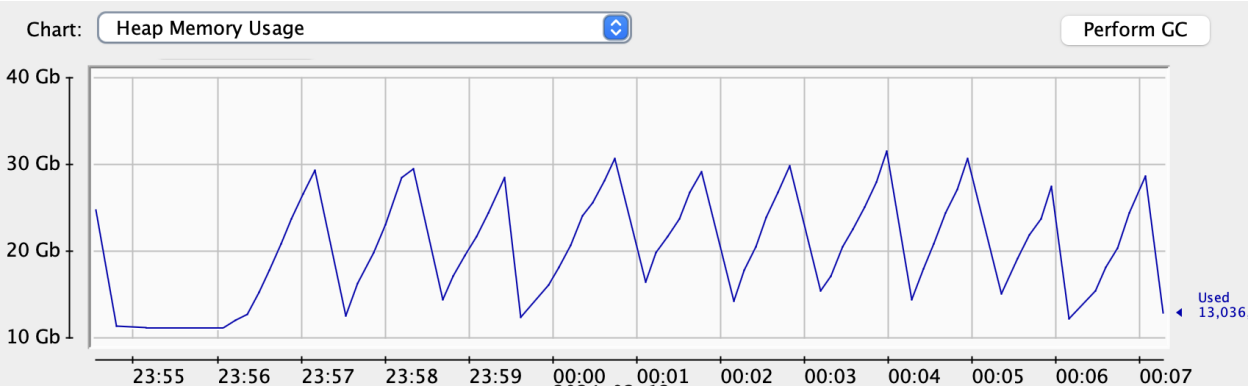
Premièrement, nous regardons un serveur cloud Intel Xeon Gold (Skylake) avec 8 coeurs à 2,3 GHz et 32 Gio de mémoire vive. Nous observons 1600 requêtes sur une durée d'environ 15 minutes. La quantité de clients simultanés monte progressivement de zéro à seize, donc de 0% à 200% de la quantité de cœurs de calcul. Le temps de réponse et le débit de requêtes sont agrégés et résumés par fenêtre de 20 secondes. Les tests ont été lancés deux fois de suite pour donner au compilateur juste-à-temps l'opportunité d'optimiser le code émis. Les résultats bruts du deuxième test sont disponibles dans les dépôts Git cités ci-dessus. Nous obtenons les graphiques suivants, qui révèlent deux régimes de fonctionnement :

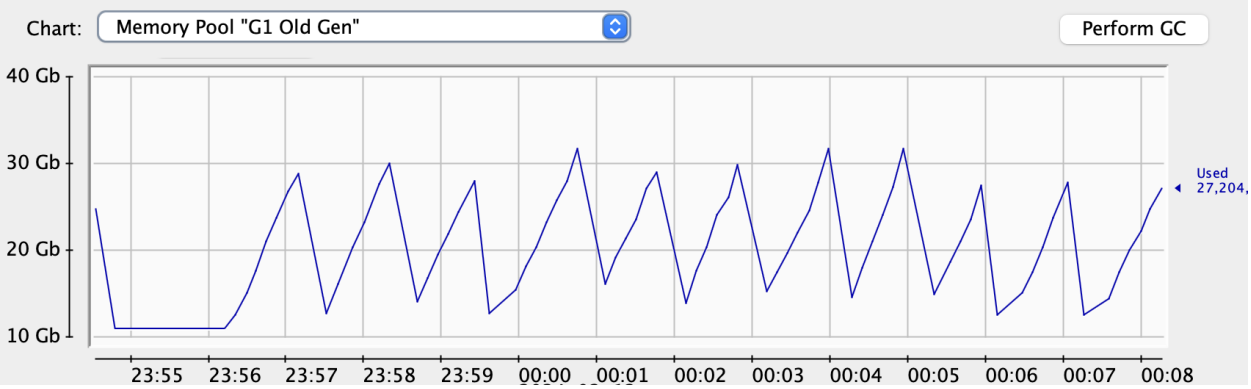


Temps de réponse et débit de requêtes en fonction de la quantité de clients simultanés.

Nous constatons une scalabilité quasi-linéaire de zéro à sept cœurs, mais avec un débit moyen assez décevant de 0,5 requête par seconde par cœur. Au-delà de sept clients simultanés, le débit chute et les temps de réponse augmentent de façon si sévère que l'ajout de serveurs supplémentaires devient nécessaire pour absorber l'excédent de charge. Nous verrons dans la prochaine section si la progression linéaire du débit se poursuit sur des instances avec plus de

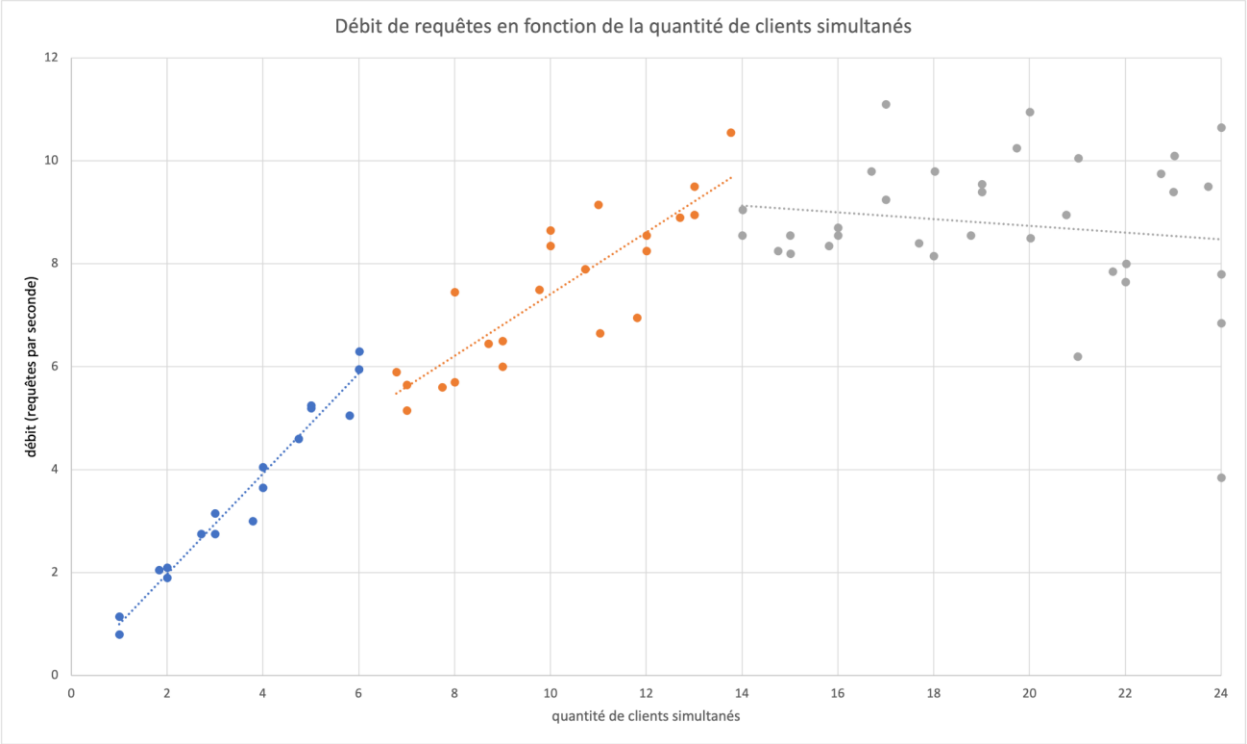
cœurs de calcul. Le plafonnement du débit à sept clients au lieu de huit est sans doute attribuable à une activité de ramassage (GC) élevé qui occupe un cœur.
Nous avons employé les Java Management Extensions (JMX) pour observer l'état interne de la machine Java distante avant les tests, puis pendant la phase la plus intense des tests avec les 8 cœurs occupés, puis surchargés. La consommation de mémoire en état d'inactivité était de 11,3 Gio. En charge lourde, les pics remontent à 30 Gio en dent de scie, avec des oscillations d'une ampleur d'environ 19 Gio qui ont lieu majoritairement dans la "vieille génération", ce qui implique un certain coût de duplication de données et de ramassage. Le serveur serait capable de fonctionner avec un tas de 30 Gio environ, ce qui correspond bien à l'offre cloud actuelle qui comporte des instances à 32 Gio de mémoire vive. Ceci dit, vu la localisation et la nature de l'activité de ramassage, un tas plus grand pourrait libérer un peu de capacité de calcul.

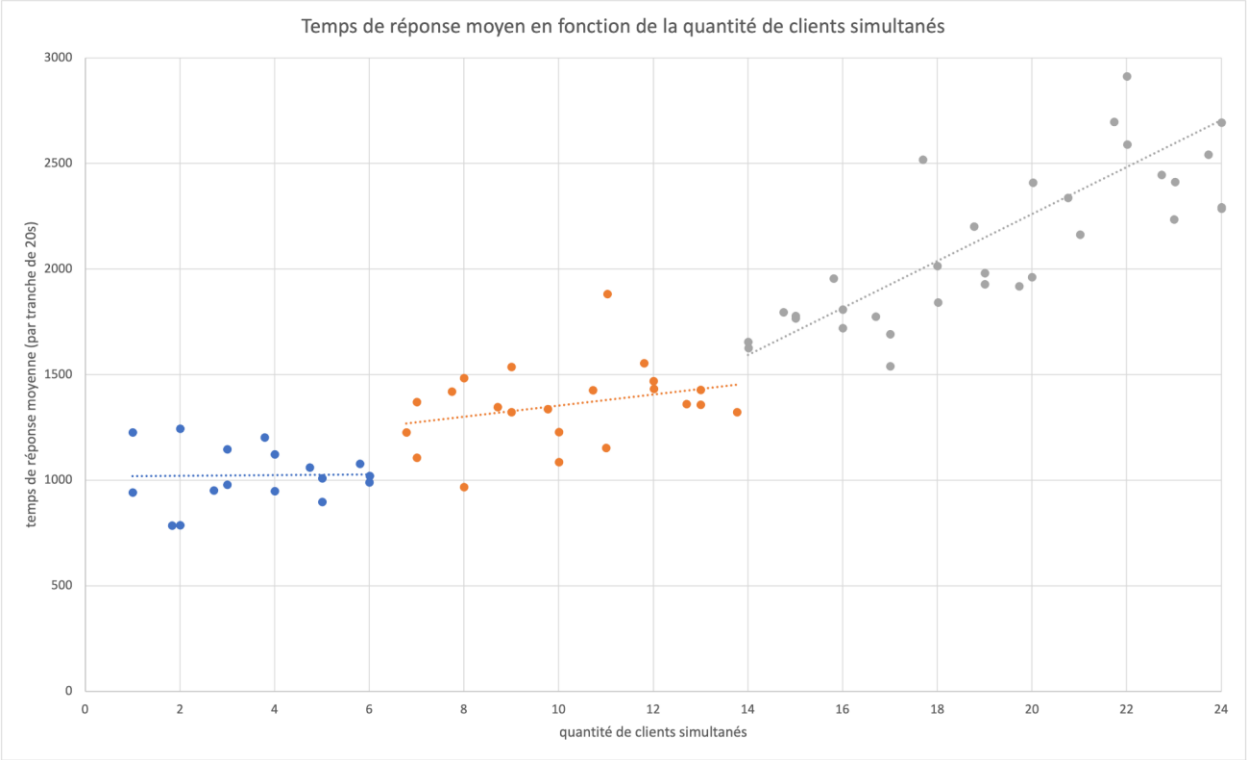




Serveur B

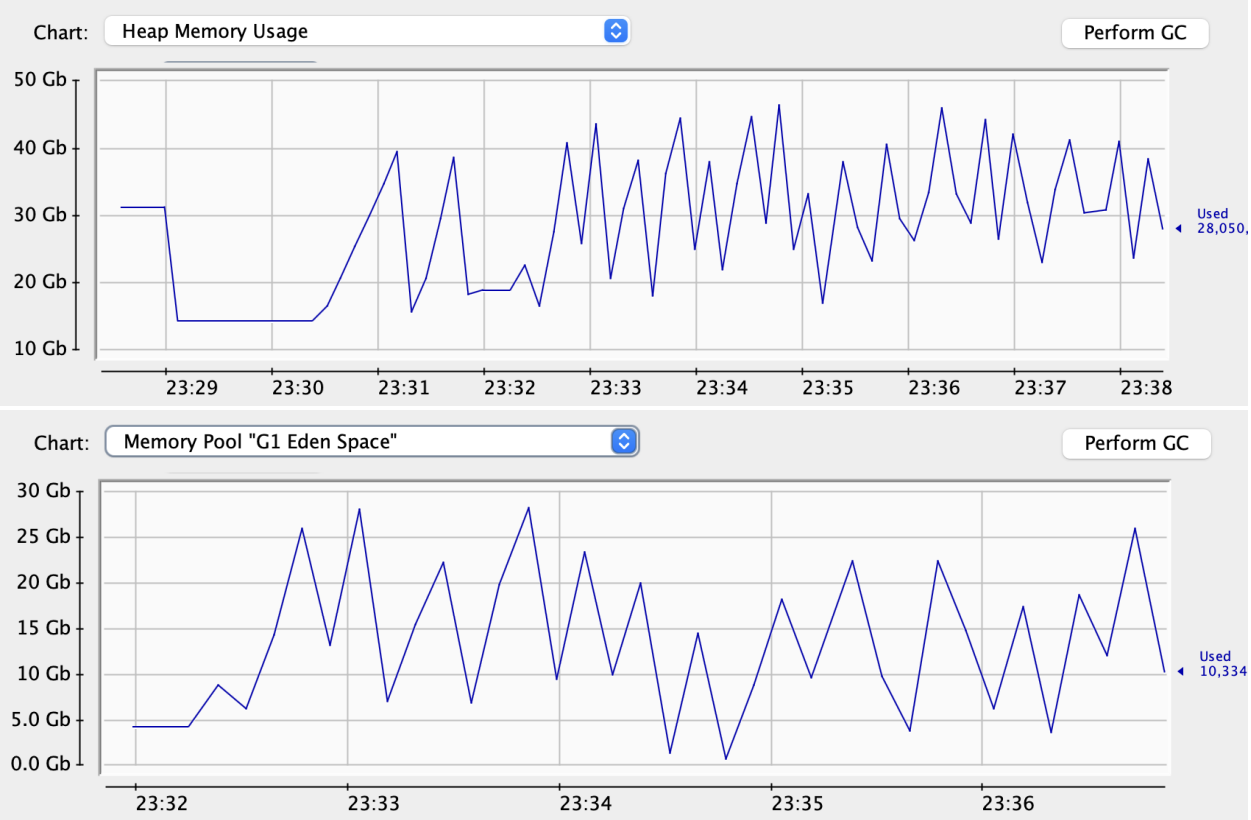
Ensuite nous regardons un serveur AMD EPYC Milan avec 16 coeurs (dédiés) à 2,4 GHz et 64 Gio de mémoire vive. La méthode est identique, mais la quantité de clients simultanés monte progressivement de zéro à vingt-quatre, donc de 0% à 150% de la quantité de cœurs de calcul. Nous observons environ 9000 requêtes sur une durée de 24 minutes. Les graphiques ainsi obtenus révèlent trois régimes de fonctionnement :





Avec ce matériel, dans le régime non saturé, le temps de réponse est divisé par deux par rapport au serveur A. Le débit augmente de façon linéaire en maintenant ces temps de réponse, pour un débit moyen d'une requête par seconde par cœur. Mais ces chiffres ne progressent pas à la même vitesse au-delà de six clients simultanés. La pente d'une approximation linéaire de la

croissance baisse d'environ 50% quand nous activons sept à quatorze clients. Si les premiers six cœurs s'occupent de six requêtes par seconde, les six prochains ne s'occupent que de quatre. Encore une fois, nous avons employé JMX pour observer la consommation de mémoire sur la machine distante, qui était comparable à celle du serveur A en état d'inactivité. Les pics en charge lourde remontent à 45 GiO en dent de scie, expliqués presque entièrement par l'activité "eden" à coût quasi-nul de ramassage. Il est donc possible que les performances ne soient pas fortement dégradées avec un tas contraint à 30 GiO ou même moins.

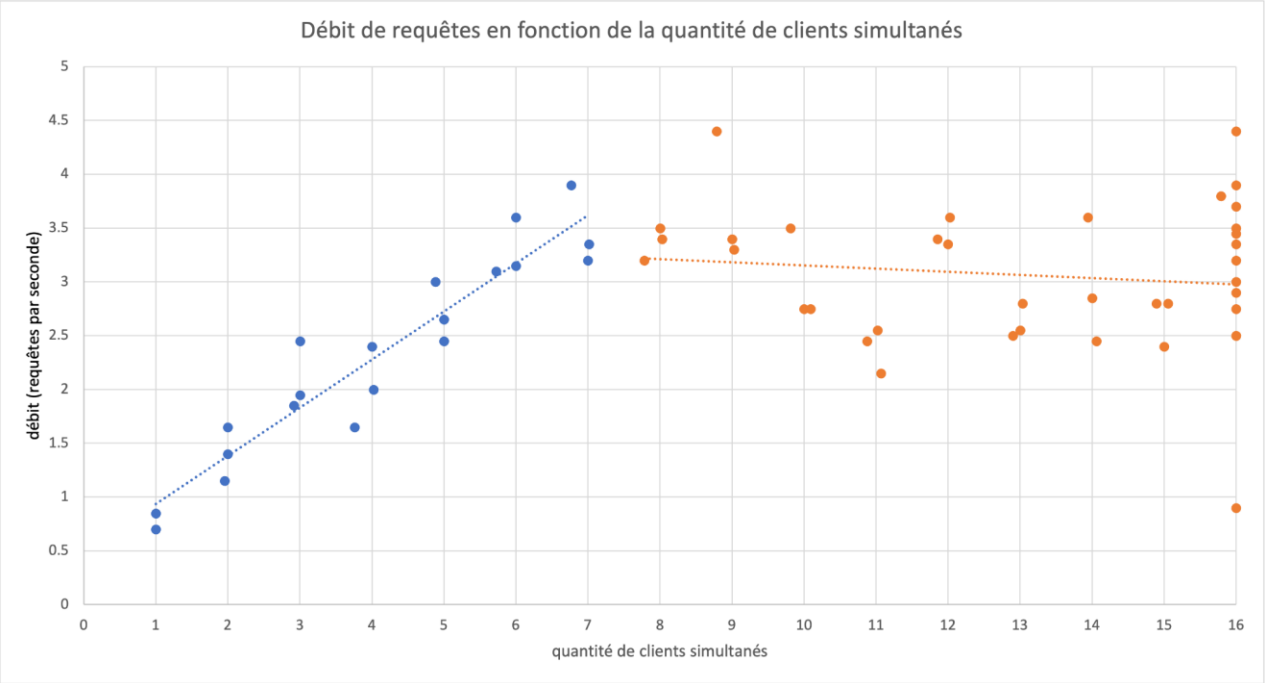


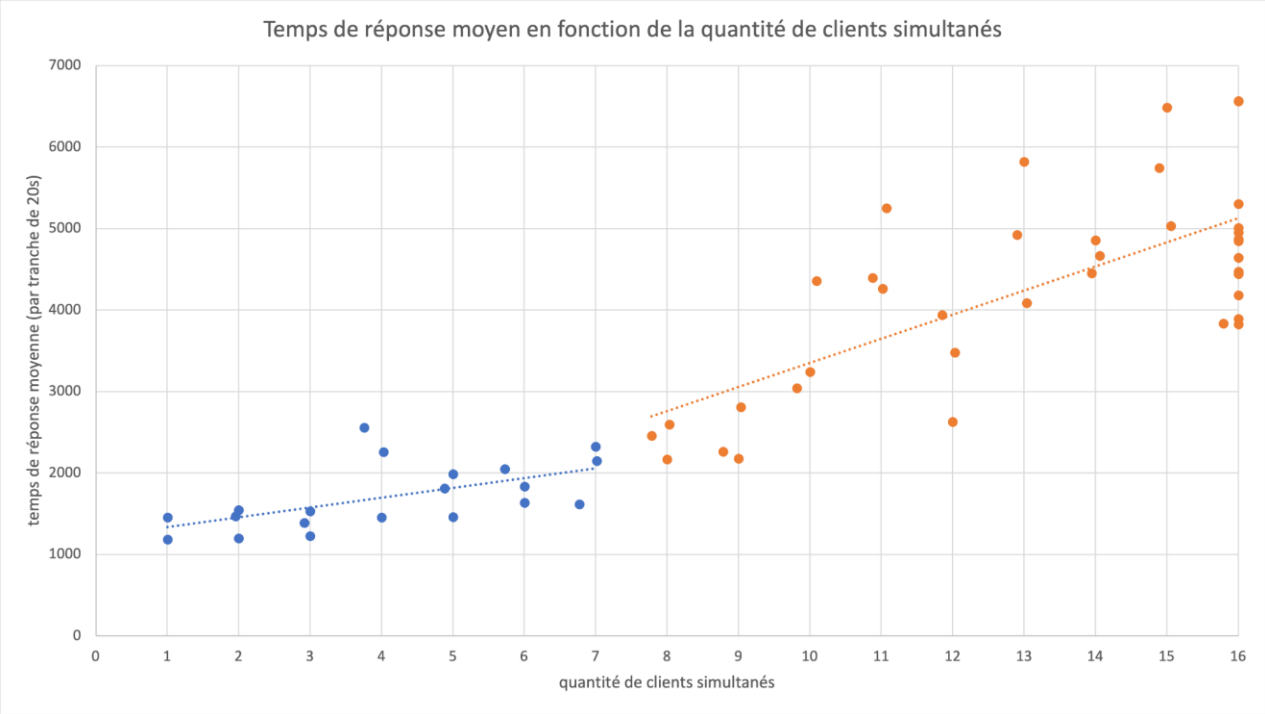
Il faudrait creuser pour déterminer si la scalabilité sub-linéaire de cette machine est due à l'architecture ou la configuration de ce matériel en particulier, ou bien si elle se reproduit sur d'autres serveurs, à cause d'une saturation du bus mémoire ou une activité GC excessive par exemple. Le fait que chaque requête est traitée deux fois plus rapidement sur ce matériel peut nous amener plus rapidement à ces états de saturation. Si ce comportement sub-linéaire est reproduisible ailleurs, il est sans doute préférable de limiter la mise à l'échelle verticale à huit cœurs par serveur, puis procéder à une mise à l'échelle horizontale avec la charge distribuée à plusieurs serveurs à huit cœurs derrière un équilibreur de charge.

Serveur C

Pour avoir des idées plus claires sur la scalabilité horizontale, le serveur C utilise exactement le même matériel que le serveur B, mais avec la moitié des ressources : une machine virtuelle avec 8 coeurs et 32 Gio de mémoire vive, avec un prix 50% moins cher. Notons que si deux instances

de type C sont allouées à une même machine physique, les performances ne seront pas forcément différentes du B. Nous aurons essentiellement la même situation mais avec deux copies des mêmes données dans deux machines virtualisées.
Avec un tas JVM de 30 Gio (et sans swap) le mémoire était épuisé et la JVM tuée par le système d'exploitation. Nous avons dû paramétrer un tas JVM de 28 Gio.



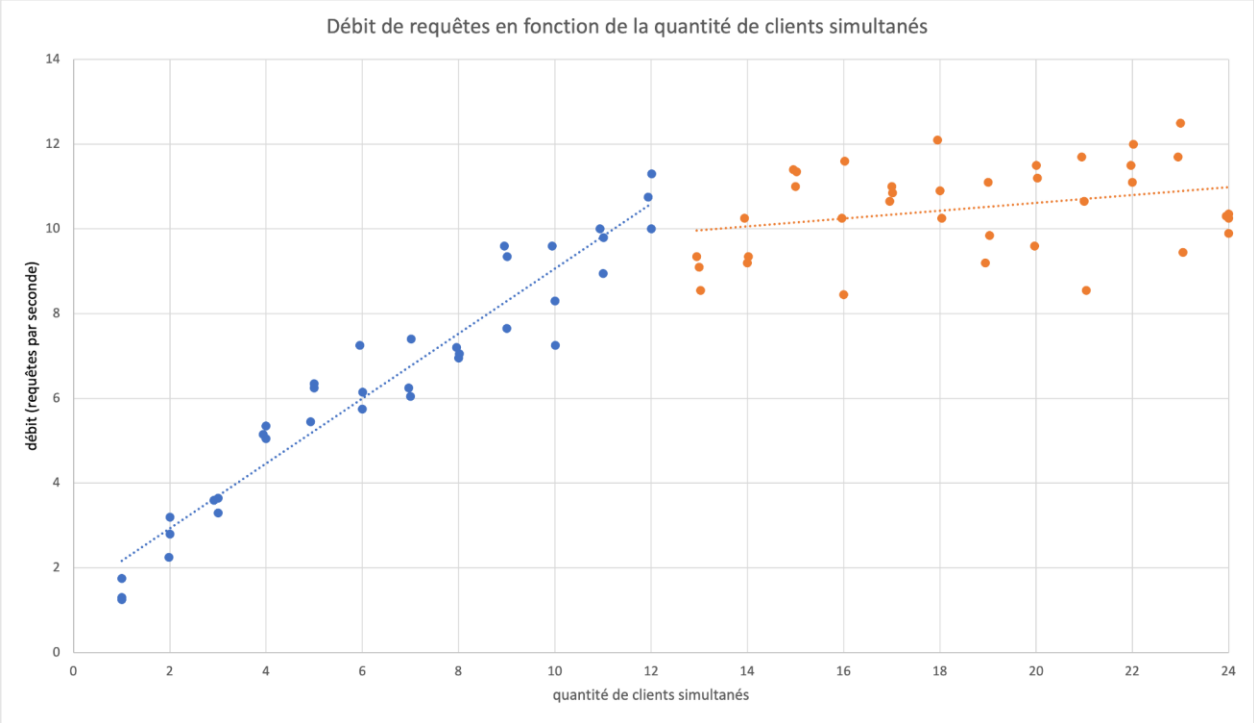


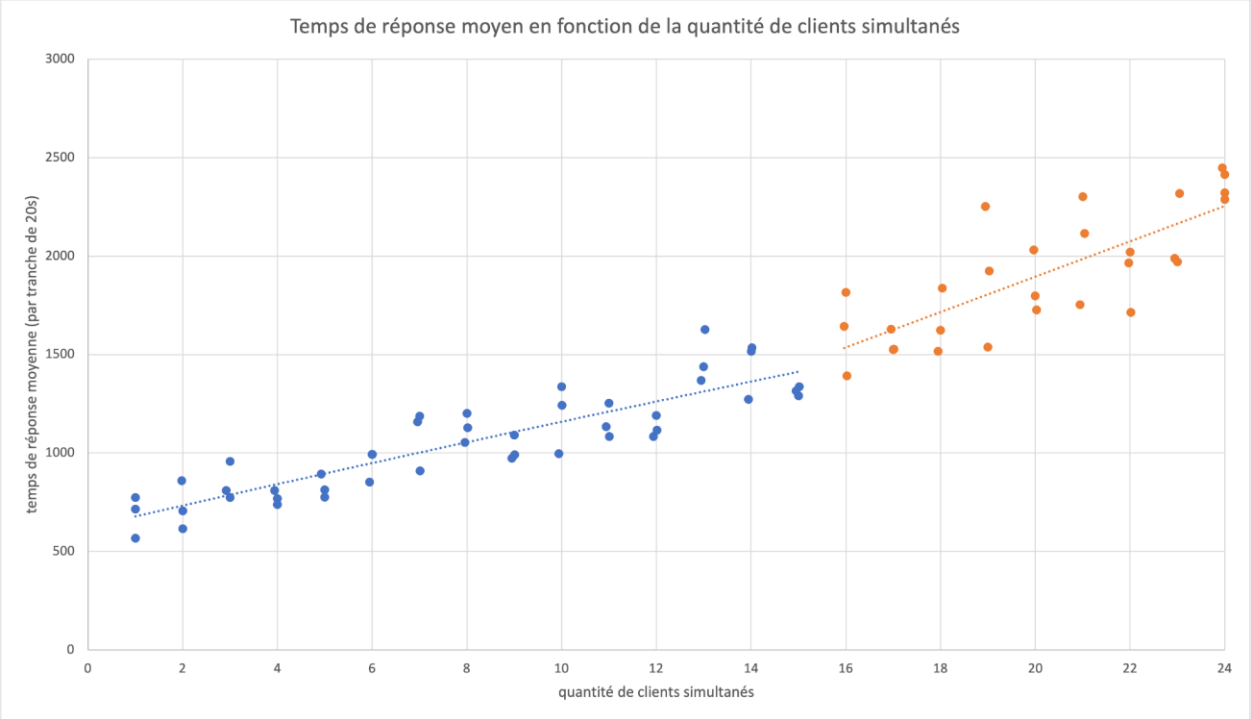
Si les performances avec un seul client correspondent approximativement à celles du serveur B, les résultats convergent rapidement vers les performances bien inférieures du serveur A quand la quantité de clients augmente. C'est un résultat intéressant : la quantité de mémoire vive

consacrée à la JVM semble être une variable importante, peut-être plus pertinent que le matériel lui-même dans les trois cas considérés.

Serveur D

AMD 7950X3D (non virtualisé) avec 16 coeurs et 128 Gio de mémoire vive.





Le temps de réponse moyen reste en dessous d'une seconde jusqu'à six clients simultanés. La croissance du débit est uniforme avec une pente légèrement en dessous d'un, jusqu'à 12 clients simultanés. Globalement comparable au serveur B mais avec un comportement un peu plus régulier et linéaire. C'est cette option qui a été retenue pour le serveur de démonstration.

Résumé et recommandations

À ce stade, avec une configuration approximative du logiciel OTP, nos impressions sur le matériel nécessaire sont les suivantes : le système production pourrait être composé d'instances employant du matériel récent, avec huit à seize cœurs et 32 à 64 Gio de mémoire vive chacune. Ces instances seraient placées derrière un équilibreur de charge qui maintient un flux d'environ huit à dix requêtes par seconde (qui représente neuf à onze clients simultanés) vers chaque instance OTP. La quantité d'instances nécessaire serait donc approximativement égale au débit pic anticipé (en requêtes par seconde) divisé par neuf. Des instances avec 16 cœurs dédiés et 64 Gio peuvent se procurer à un prix d'environ 100 EUR / mois minimum. Le coût mensuel du service d'hébergement serait donc au minimum 1000 EUR par mois, par tranche de débit de 100 requêtes par seconde.

Remarques qui sortent de l'étude demandée

En dehors de l'étude spécifique d'OpenTripPlanner, nous souhaitons ajouter quelques éléments qui nous semblent importants à prendre en compte dans le cadre d'un calculateur national.

Collecte des conditions tarifaires

La normalisation de la représentation des différents tarifs et leurs conditions est un travail à ne pas sous-estimer, d'autant plus qu'il y a assez peu de tarifs publiés actuellement et d'outillage.
Si le système visé souhaite faire du post-paiement, en optimisant le tarif selon le nombre de trajets (par exemple basculer sur un ticket journalier au bout de 4 trajets dans la journée) et selon le profil (tarif étudiant), alors il faudra avoir une base de tarifs qui dépasse le simple ticket unité.
Il serait opportun de travailler avec les start-ups de beta.gouv.fr qui travaillent sur l'accès aux droits, dont les tarifs sociaux des transports en commun sont une composante importante.

Compatibilité de licence

La licence OdBL utilisée par la majorité des jeux de données n'est pas compatible avec la Licence Mobilité, car cette dernière impose un droit de regard sur la manière d'utiliser la données.
Étant donné qu'il s'agit de données de même nature (horaire de transport en commun), de même granularité (données théorique pour un futur proche de quelques semaines) et de même zone géographique avec des chevauchements.
Le calculateur national ne pourra donc pas intégrer les données qui sont publiées sous la licence mobilité.
Pour les données en Licence Ouverte, il n'y a pas de restriction à les intégrer dans des jeux de données contenant des données en OdBL ou licence mobilité.

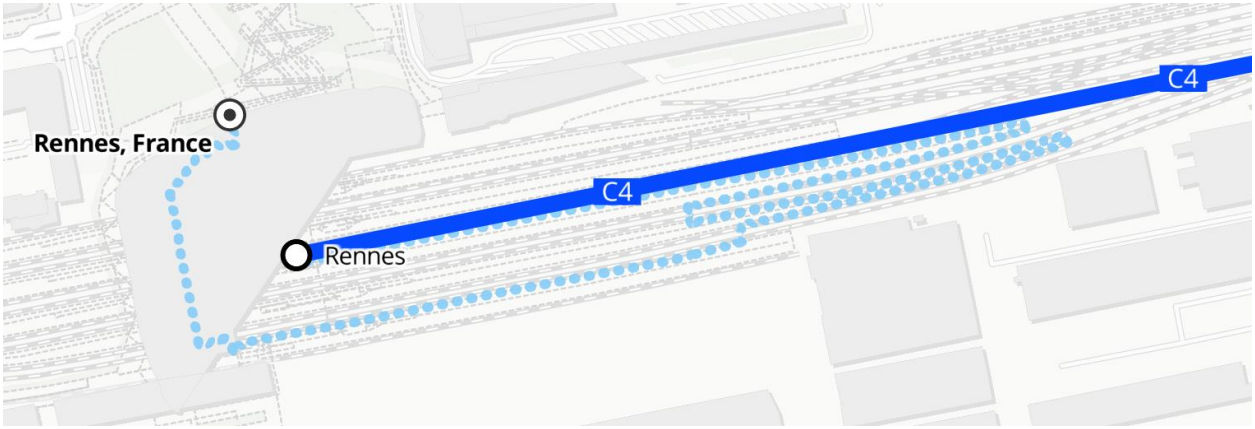
Le temps réel n'est pas concerné par ces restrictions, car il ne s'agit pas de la même granularité des données.

Base nationale des arrêts

Gestion des correspondances

Même si OpenStreetMap permet de trouver des cheminements piétons dans les ensembles complexes tels que les gares, il faudra s'assurer d'une mise en qualité pour éviter des durées de correspondances

- trop longues : cas ci-dessous à Rennes
- trop courtes : cas présenté auparavant pour une correspondance train-métro situés géographiquement sur le même point, mais avec un cheminement complexe



Ce sont typiquement les gares qui vont être les endroits où se feront les correspondances entre deux réseaux.

Nommage des arrêts

À la gare de Rennes, selon les jeux de données, l'arrêt peut s'appeler :

- Rennes (jeu de données SNCF TGV)
- Gares (jeu de données urbain SNCF)
- Gare Routière (jeu de données lignes départementales)
- Gare de Rennes (jeu de données TER Normandie)

Cette duplication rend difficile la sélection par nom d'arrêt et rend peu lisible l'offre disponible à un arrêt donné.

Publication de ces tableaux de correspondance

L'entité qui réalisera le calculateur d'itinéraire devra donc générer des tableaux de correspondances entre arrêts de divers jeux de données (que ce soit pour indiquer qu'il s'agit du même arrêt, ou proposer des durées réalistes).
Il serait très souhaitable que l'appel d'offre stipule que ces données soient publiées en Open Data pour permettre l'amélioration continue de l'ensemble de la qualité de l'ensemble des calculateurs existants.