

Mémoire professionnel de 3^{ème} année

présenté par

Othmane CHAOUCHAOU

Science de numérique - Système Logiciel

Années 2020-2023

Grid Computing Efficiency

Stage de fin d'étude chez



Tuteur de l'entreprise : **Hani SEIFEDDINE**, Senior principal engineering lead

Tuteur de l'école : **Xavier CRÉGUT**, Professeur à INPT/ENSEEIH

Août 2023

ENSEEIH - École Nationale Supérieure d'Électrotechnique, d'Électronique,
d'Informatique, d'Hydraulique et des Télécommunications

2 Rue Charles Camichel, 31000 Toulouse

Tél. +33 (0)5 34 32 20 00

ENSEEIH

All that is gold does not glitter
Not all those who wander are lost
The old that is strong does not wither
Deep roots are not reached by the frost
From the ashes a fire shall be woken
A light from the shadows shall spring
Renewed shall be blade that was broken
The crownless again shall be king

Remerciements

Je tiens tout d'abord à remercier l'ensemble des personnes chez Murex que j'ai eu l'occasion de côtoyer au cours de ce stage et qui ont contribué à son bon déroulement.

Je tiens à remercier en premier lieu mon maître de stage **Hani SEIFEDDINE**, Senior principal engineering lead à Murex SaS, pour avoir supervisé mon stage et encadré mon travail. Je le remercie pour son temps, son accompagnement et pour les échanges que nous avons eus.

Je tiens à exprimer ma profonde gratitude envers toute l'équipe de Murex pour leur accueil chaleureux. Chaque rencontre et échange avec les membres de cette entreprise ont été précieux. Une mention spéciale pour les autres stagiaires avec qui j'ai partagé cette expérience ; leur camaraderie et leurs retours ont considérablement enrichi mon passage chez Murex.

Finalement, je souhaite dédier une pensée toute particulière à mes parents et à mes frères et sœurs. Leur soutien inébranlable, leurs encouragements et leur confiance en moi ont été les piliers sur lesquels j'ai construit mes aspirations. Je leur dois énormément, car leurs sacrifices et leurs conseils ont façonné mon parcours. Leur amour et leur compréhension ont été une source constante de force, me poussant à rechercher l'excellence dans tout ce que j'entreprends.

Résumé

Ce stage se déroule au sein de l'éditeur de logiciels pour la finance Murex, dans l'équipe MSA Platform Architecte, faisant partie intégrante du département du développement de Murex. Il s'articule autour d'un projet d'exploration pour élaborer un prototype du service de gestion du risque de Murex, sans recourir à l'outil de Murex lui-même.

L'objectif est double : d'une part fonctionnel, où il est question de s'immerger dans le monde de la finance de marché, avec une focalisation sur la calibration et le stress test. Cette immersion vise à reproduire le service de gestion du risque en utilisant des technologies open source, en particulier l'API ORE, tout en contournant les contraintes propres à Murex. D'autre part, l'aspect technique implique le déploiement du service en mode distribué tout en exploitant la technologie de cache Redis pour optimiser les performances.

Cette mission, riche en dimensions (technique, organisationnelle, humaine et fonctionnelle), offre une opportunité d'appliquer l'ensemble des compétences acquises durant ma formation faite à l'ENSEEIH, notamment celles liées à ma spécialisation en système logiciel.

Table des matières

Glossaire	3
Acronymes	4
1 Introduction	5
2 Contexte du stage	6
2.1 Présentation de l'entreprise	6
2.1.1 Murex	6
2.1.2 Les Valeurs de l'entreprise	6
2.1.3 Département et Équipes	7
2.2 Présentation de la problématique	7
2.3 Enjeux	8
2.3.1 Fonctionnel	8
2.3.2 Technique	8
2.4 Présentation des missions réalisées	8
3 Missions réalisées	10
3.1 Service de Calibration et de Stress des Paramètres de Marché	10
3.1.1 Finance de marchés	10
3.1.1.1 Instruments de la finance du marché	11
3.1.1.2 Trade	11
3.1.1.3 Market Data	12
3.1.2 Gestion de risque	12
3.1.2.1 Calibration	13
3.1.2.2 Stress Testing	14
3.1.3 Bibliothèques de pricing	16
3.1.3.1 Open Gamma Strata	16
3.1.3.2 Open Source Risk Engine	18
3.1.4 Prise en main d'ORE	21
3.1.4.1 Première expérience	21
3.1.4.2 Deuxième expérience	23
3.1.5 Engin de calcul	28
3.1.5.1 Architecture	28
3.1.5.2 Générateur de trades	28
3.1.5.3 Générateur de scénarios	29
3.1.5.4 Stress Tester	30
3.1.5.5 Résultats du calcul séquentiel	30
3.2 Application de la technologie du cache au service	31
3.2.1 Redis	31

3.2.1.1	Origines et évolution historique de Redis	32
3.2.1.2	Fonctionnement interne de Redis	32
3.2.2	Mise en œuvre de Redis dans notre système - Première tentative	35
3.2.2.1	Vue d'ensemble de la 1 ^{ère} Architecture	35
3.2.2.2	Client Redis	35
3.2.2.3	Diagramme UML du Client	35
3.2.2.4	Sérialisation des Trades	38
3.2.2.5	Lua Script	38
3.2.2.6	ORE et son interaction avec Redis	39
3.2.3	Mise en œuvre de Redis dans notre système - Deuxième tentative	40
3.2.3.1	Vue d'ensemble de la 2 ^{ème} Architecture	41
3.2.3.2	Store	41
3.2.3.3	Task Trigger	41
3.2.3.4	Distribute	41
3.2.3.5	Get&Delete	42
3.2.4	Diagramme de Séquence	42
3.2.5	Déploiement de l'architecture	43
3.2.5.1	Engins	43
3.2.5.2	Redis	43
3.3	Résultats	44
3.3.1	Tests	44
3.3.2	Synthèse et considérations futures	45
3.3.2.1	Utilisation d'un cluster Redis	45
3.3.2.2	Utilisation du Machine Learning	46
4	Conclusion	47
4.1	Conclusion fonctionnelle	47
4.2	Conclusion technique	47
4.3	Conclusion personnelle	48
	Table des figures	49
	Liste des tableaux	50
	Liste des codes	51
	Bibliographie	52
	Table des annexes	53

Glossaire

Cluster En informatique, un cluster désigne un ensemble de machines indépendantes travaillant conjointement pour former une entité unique et performante. Cette mise en commun permet d'obtenir des performances et une disponibilité supérieures à celles d'une machine unique, tout en assurant une meilleure répartition des charges et une tolérance aux pannes. 2

Désérialisation Processus de traduction d'un tableau de bytes pour le reconvertir à son objet de programmation original . 2

Sérialisation Processus de traduction d'un objet de programmation en un tableau de bytes . 2

XML Le XML, ou eXtensible Markup Language, est un langage de balisage qui permet de définir et de contrôler le contenu de documents. Il est principalement utilisé pour faciliter l'échange de données entre différentes applications ou systèmes. Contrairement à HTML qui a un ensemble fixe de balises destiné à la présentation des données, XML permet aux utilisateurs de définir leurs propres balises, rendant ainsi le langage extensible et adaptable à une multitude de besoins. 2

Acronymes

ORE Open source Risk Engine. 2

CHAPITRE 1

Introduction

Le travail actuel fait partie de mon projet de fin d'étude dans le cadre de ma formation d'ingénieur en informatique et télécommunication à l'ENSEEIH. Plusieurs raisons ont motivé ma décision de postuler pour cette offre proposée par Murex SaS.

Tout d'abord, j'aspirais à acquérir une expérience significative dans le domaine de la finance et du développement logiciel.

Comme chaque nouvelle expérience, ce stage représentait un véritable challenge pour moi. La nécessité d'allier créativité, autonomie et curiosité était primordiale pour la réussite de cette mission. Ce fut également une opportunité unique de me plonger dans l'univers de la finance de marché tout en l'associant à l'informatique.

Mon projet de fin d'étude était également une expérience intéressante, car le sujet était novateur et l'équipe avec laquelle j'ai travaillé traitait également de projets innovants.

Bonne lecture

CHAOUCHAOU Othmane
Mars - Septembre 2023
Murex SaS
Paris (France)

CHAPITRE 2

Contexte du stage

2.1 Présentation de l'entreprise

2.1.1 Murex

Murex est un éditeur de logiciels fondé en 1986 offrant des produits pour les activités de trading, de trésorerie, de risque et de traitement des transactions pour les acteurs des marchés financiers.

Outre le bureau principal de Paris, Murex compte 17 bureaux à travers le monde dans des villes comme New York, Londres, Dublin, Hong Kong, Beyrouth, Sydney et Singapour.

Les clients de Murex se trouvent dans 70 pays. L'entreprise a connu une croissance organique et emploie aujourd'hui environ 2600 salariés dans le monde, dont 900 à Paris.

Depuis sa création, Murex a joué le rôle de leader dans son secteur en proposant à tout acteur une technologie efficace permettant de croître et innover dans le secteur financier.

La finance est un secteur très diversifié où existent de nombreux processus complexes. La compagnie a dévoué plus de 30 ans au design, à l'implémentation et l'évolution des activités de trading, du management du risque pour ses clients, la gestion des processus, et les opérations ayant lieu après trading.

Tout ceci a mené à MX.3[1], 3ème génération de la plate forme. Celle-ci, est utilisée par de nombreux acteurs du monde de la finance. Parmi ses clients, on retrouve des banques, des gestionnaires d'actifs, des fonds de pension et des compagnies d'assurances.

2.1.2 Les Valeurs de l'entreprise

Aujourd'hui, Murex est le troisième éditeur français de logiciel avec un chiffre d'affaires estimé à environ 460 millions d'euros. De nombreux nouveaux acteurs ont vu le jour au fil des années mais la compagnie reste leader de son marché.

On retrouve à l'origine de ce succès principalement deux valeurs :

- • **Innovation** : Sur les brochures, logos, sites, apparait le slogan historique "PIONEERING AGAIN" qui pourrait se traduire comme "innover à nouveau". L'éditeur cherche constamment à améliorer sa technologie et ses outils.

- • **Partenariat client** : Collaboration continue avec les clients en jouant le rôle de conseiller de confiance, ainsi que fournisseur d'un outil facilitant leurs désirs de transformation et de croissance.

Murex a, au travers de son désormais conséquent historique de projets, prouvé qu'il supporte tout type de clients, de toute taille. Et aujourd'hui, plus de 45 000 utilisateurs à travers le monde manipulent l'outil quotidiennement pour réaliser les activités qui sont spécifiques à leur cœur de métier.

Son savoir-faire et ses services de haute qualité lui ont permis de remporter à plusieurs reprises ces dernières années la place de premier fournisseur de technologie du secteur. Ces distinctions sont accordées par des analystes et sont basées sur des votes et/ou sélections de jurys. Murex a par exemple été votée comme premier fournisseur technologique consécutivement pour les 5 dernières années par les lecteurs d'Asia Risk et Risk Magazine, preuve du rayonnement au-delà des frontières françaises ou bien même européennes.

2.1.3 Département et Équipes

Mon stage s'est déroulé dans l'équipe MSA Platform Architecte, une équipe de département du développement.

En dehors de ce département, il en existe plusieurs autres. J'ai principalement interagi avec une d'entre eux durant mon stage, à savoir **PES**. J'ai également été en relation avec **is-dev-ops** qui m'a aidé à résoudre un problème de build en milieu de stage.

Département	Description
DEV	C'est l'équipe dont la mission est le développement du logiciel MX.3.
Product Evolution Services (PES)	Cette équipe est responsable de l'évolution de la plateforme. Elle s'assure que les solutions développées sont en accord avec les besoins du marché (technologies, besoin client).
Quality Assurance (QA)	L'équipe est chargée de s'assurer de la bonne qualité du logiciel.
Client Evolution and Support (CES)	Les CES sont des consultants dont la mission est d'accompagner au quotidien des clients de Murex en leur fournissant des services.
Customer Delivery Service (CDS)	Cette équipe est en charge de l'implémentation et des livraisons des solutions proposées par la plateforme.
Release Management (RM)	Cette équipe gère les nombreuses versions du logiciel ainsi que le processus de release.

2.2 Présentation de la problématique

L'univers de la finance de marché est en constante évolution, ce qui exige une adaptation constante des outils et méthodes utilisés pour naviguer dans ce domaine complexe. Parmi les défis majeurs auxquels est confronté ce secteur figurent la gestion du risque et la mise en œuvre efficace des architectures distribuées pour les moteurs de calcul.

La nécessité d'innover et de rester compétitif dans ce secteur exige une réévaluation constante des méthodes traditionnelles de gestion du risque. Les fluctuations du marché, les changements dans la législation et la réglementation financière, et l'évolution des produits financiers sont autant de facteurs qui nécessitent une approche de gestion du risque efficace, précise et adaptable.

Parallèlement, l'application des architectures distribuées dans les moteurs de calcul offre de nouvelles possibilités en termes de capacité de calcul et d'efficacité. Cependant, leur mise en œuvre soulève des questions complexes en termes de gestion des données, d'optimisation des performances et de sécurité.

Une problématique clé qui se pose est donc : Comment utiliser efficacement des outils open source pour émuler l'environnement Murex, une plate-forme financière leader dans la gestion du risque, tout en gérant ces défis ? Comment ces outils peuvent-ils être adaptés pour répondre aux exigences changeantes du secteur financier ? Et comment peuvent-ils être utilisés pour optimiser l'utilisation des architectures distribuées dans les moteurs de calcul ?

2.3 Enjeux

2.3.1 Fonctionnel

Le premier enjeux de cette étude est purement fonctionnel. Il s'agit de comprendre le fonctionnement de la finance du marché, afin de chercher l'outil adéquat pour simuler la gestion de risques dans Mx. Ensuite, il faut utiliser cette outil pour créer un engin qui fait la calibration et le stress test en mode séquentiel.

2.3.2 Technique

Le deuxième enjeux est d'utiliser une technologie pour améliorer les performances de l'engin créée. Ensuite, il faut concevoir une architecture en mode distribuée qui utilise l'engin. Parmi les technologies, on envisagera le caching.

2.4 Présentation des missions réalisées

Durant mon stage chez Murex, ma mission principale consistait à élaborer un prototype opérationnel pour un service nouvellement envisagé. Au sein de cette entreprise, j'ai bénéficié d'une position privilégiée en tant que stagiaire, me permettant d'échanger régulièrement avec des équipes à la fois techniques et fonctionnelles. Cette interaction régulière m'a conféré une compréhension approfondie des exigences inhérentes au projet. Elle m'a également permis de participer activement à des discussions stratégiques concernant les solutions à mettre en œuvre. Par conséquent, cette mission s'est avérée être une opportunité inestimable pour moi, m'offrant une immersion pratique dans le développement de solutions novatrices face aux défis contemporains de la finance de marché.

Au sein de Murex, les missions s'articulaient autour de cinq grands axes :

- • La prise en main des API Open Source de calculs des risques de marchés.
- • Justifier l'utilisation de la bibliothèque choisie et présenter ces avantages et inconvénients .
- • La calibration et le stress testing en mode séquentiel de plusieurs produits financiers (bond, courbe, etc.) en utilisant la bibliothèque choisie.
- • Optimisation des performances par le caching en utilisant la technologie Redis.

- • La mise en place d'outils pour surveiller et analyser les problèmes fonctionnels ou techniques potentiels (tels que les problèmes de performance ou de disponibilité) dans un environnement de production.

Afin d'atteindre les objectifs tracés, il était primordial de mettre en place une méthode de travail pour une meilleure conduite de projet. Pour ce faire, le présent mémoire est organisé en deux parties. La première partie sera consacrée à la partie finance du marché, les bibliothèques financières manipulées durant le stage et la création d'un service séquentiel de calibration et de stress des paramètres de marchés. Dans la seconde partie, nous allons utiliser le cache Redis afin d'améliorer les performances et décrire l'architecture mis en place en mode distribué.

CHAPITRE 3

Missions réalisées

3.1 Service de Calibration et de Stress des Paramètres de Marché

La première partie du stage, qui a une forte composante fonctionnelle, vise à développer un service pour la gestion des risques de marché. Avant d'entrer dans le vif du sujet, il est essentiel de définir brièvement le vocabulaire financier utilisé dans ce domaine. Cela non seulement donnera au lecteur une idée du contexte fonctionnel de l'entreprise Murex, mais facilitera aussi la compréhension des bibliothèques financières qui seront utilisées tout au long de ce stage.

3.1.1 Finance de marchés

La finance de marchés est un domaine qui concerne l'achat, la vente et l'échange d'actifs financiers sur divers marchés. Les actifs concernés peuvent inclure des actions, des obligations, des devises, des produits dérivés, et bien d'autres. Il s'agit d'un domaine clé de la finance qui joue un rôle crucial dans l'économie globale, en facilitant le transfert de capital, de risque et de liquidités.

Les acteurs du marché financier comprennent à la fois des institutions financières (comme les banques, les fonds de pension et les compagnies d'assurances) et des particuliers. Les transactions sur les marchés financiers peuvent être réalisées soit sur des marchés de gré à gré (OTC), où les transactions sont réalisées directement entre deux parties sans passer par une bourse, soit sur des marchés organisés comme les bourses de valeurs.

L'un des principaux risques auxquels les acteurs des marchés financiers sont confrontés est le risque de marché, qui est le risque de subir des pertes en raison des fluctuations des prix des actifs financiers. C'est ici que le service de calibration et de stress des paramètres de marché entre en jeu, comme nous le verrons dans les sections suivantes.

3.1.1.1 Instruments de la finance du marché

Les instruments de la finance de marché sont les outils que les investisseurs utilisent pour acheter, vendre, échanger ou couvrir des actifs financiers. Ces instruments peuvent prendre de nombreuses formes, dont certaines sont :

- **Actions** : Ces instruments représentent la propriété d'une partie d'une entreprise. Les détenteurs d'actions peuvent recevoir une part des bénéfices de l'entreprise sous forme de dividendes et peuvent également bénéficier de l'appréciation du cours de l'action.
- **Obligations** : Ces instruments représentent un prêt effectué par un investisseur à un émetteur (généralement une entreprise ou un gouvernement). En retour, l'émetteur s'engage à rembourser le principal à une date d'échéance spécifique et à verser des intérêts périodiques à l'investisseur.
- **Produits dérivés** : Ces instruments tirent leur valeur d'un actif sous-jacent, qui peut être une action, une obligation, une devise, une matière première, un indice boursier, etc. Les produits dérivés sont souvent utilisés pour la couverture du risque ou la spéculation. Les exemples incluent les futures, les options, et les swaps.
- **Devises** : Ces instruments sont utilisés dans le marché des changes (Forex), où les devises sont achetées et vendues. Les fluctuations des taux de change peuvent avoir un impact significatif sur les investisseurs et les entreprises qui opèrent au niveau international.
- **Fonds d'investissement** : Ces instruments regroupent les capitaux de nombreux investisseurs pour investir dans un portefeuille diversifié d'actifs. Les fonds communs de placement, les fonds indiciels et les fonds négociés en bourse (ETF) sont quelques exemples de fonds d'investissement.

Ces instruments de marché sont essentiels dans la finance moderne et forment la base de nombreuses stratégies d'investissement et de gestion des risques.

3.1.1.2 Trade

Dans le monde de la finance, un trade désigne l'acte d'acheter et de vendre des instruments financiers. Cela peut être des actions, des obligations, des devises, des contrats à terme, des options, etc. Chaque trade implique au moins deux parties, un acheteur et un vendeur, qui s'accordent sur un prix pour l'instrument financier en question.

Techniquement parlant, un trade est constitué de plusieurs informations essentielles telles que l'identifiant du trade, le type de l'instrument financier, le volume ou le nombre d'unités de l'instrument à acheter ou à vendre, le prix de l'instrument, la date de la transaction et les parties concernées par la transaction.

Chaque trade est généralement associé à une série de cash-flows prévus, qui représentent les paiements attendus liés à l'instrument financier. Par exemple, pour un trade sur une obligation, les cash-flows prévus peuvent comprendre les paiements de coupons périodiques et le remboursement du principal à l'échéance.

Les trades sont essentiels pour les activités de trading et de gestion des risques. Ils servent de base pour la valorisation des portefeuilles, la mesure de la performance, la gestion des risques de marché, de crédit et de liquidité, entre autres.

Cependant, la gestion efficace des trades nécessite l'accès à des données de marché précises et à jour. Les prix des instruments financiers, les taux d'intérêt, les taux de change, les indices de volatilité, etc., sont autant d'informations essentielles pour la valorisation des trades, l'évaluation des risques et la prise de décision. C'est ici que les Market Data entrent en jeu, permettant une gestion efficace et précise des trades.

3.1.1.3 Market Data

Les données de marché, ou "Market Data", constituent l'ensemble des informations concernant les conditions de marché pour les instruments financiers et les matières premières. Celles-ci incluent des informations telles que les prix courants, les taux d'intérêt, les taux de change, les indices de volatilité, etc. Ces données sont essentielles pour de nombreuses opérations financières, y compris le trading et la gestion des risques.

En effet, comme mentionné précédemment dans la section "Trade", la gestion des trades nécessite un accès à des données de marché précises et à jour. Sans des données de marché fiables, il serait difficile, voire impossible, d'évaluer correctement la valeur des trades ou de mesurer les risques associés.

Sur le plan technique, les données de marché peuvent être obtenues à partir de plusieurs sources, y compris les bourses, les fournisseurs de données financières, et même directement à partir des contreparties de trading. Ces données sont généralement stockées et gérées à l'aide de systèmes de gestion de données de marché, qui permettent aux utilisateurs de rechercher, d'accéder et d'utiliser les données de manière efficace.

Cependant, la gestion des données de marché peut être un défi en soi. Non seulement ces données doivent être constamment mises à jour pour refléter les conditions actuelles du marché, mais elles doivent également être nettoyées et transformées en un format utilisable, ce qui peut nécessiter des traitements importants. De plus, les données de marché peuvent souvent être volumineuses et complexes, nécessitant des systèmes de stockage et de gestion des données performants et robustes.

C'est dans ce contexte qu'interviennent la calibration et le stress testing des paramètres de marché. Ceux-ci permettent de modéliser les fluctuations possibles du marché et leur impact potentiel sur les trades, contribuant ainsi à une gestion efficace des risques.

3.1.2 Gestion de risque

La gestion des risques est un processus méthodique et structuré qui nécessite une attention méticuleuse à chaque étape. Après avoir établi un trade, il est crucial de rassembler les données de marché nécessaires qui façonneront les décisions prises plus tard, car ces données fournissent le contexte dans lequel le trade opère. Une fois ces données en main, la phase de calibration commence. Le stress testing suit la calibration, et le trade est validé à la fin.

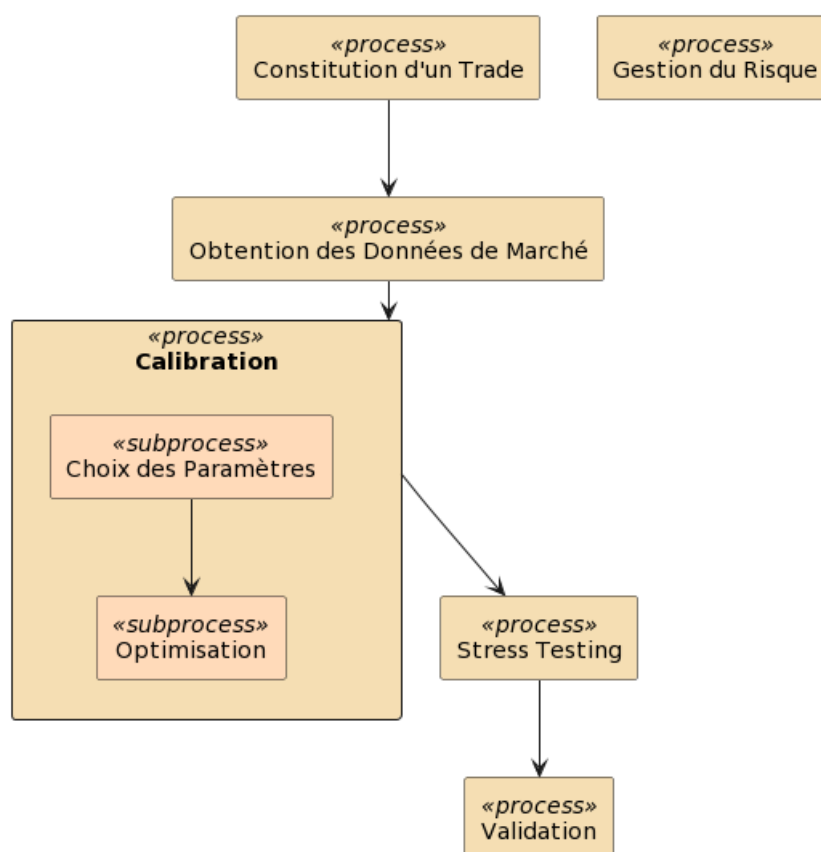


FIGURE 3.1.1 – Étapes de la gestion du risque d'un trade

3.1.2.1 Calibration

La calibration est un processus essentiel qui fait suite à la constitution d'un trade, l'obtention des données de marché pertinentes, et précède le stress testing. Ce processus consiste à ajuster les paramètres d'un modèle financier afin que le prix de l'instrument financier dans le modèle coïncide le plus étroitement possible avec le prix observé sur le marché. Cela permet de garantir que le modèle reproduit fidèlement la réalité du marché et offre ainsi une base solide pour la gestion du risque.

Techniquement, la calibration consiste à résoudre un problème d'optimisation. Pour chaque produit financier ou pour chaque type de produit financier, un certain nombre de paramètres du modèle sont choisis. Les valeurs de ces paramètres qui minimisent l'écart entre les prix du modèle et les prix observés sur le marché sont ensuite déterminées. Ce processus peut être complexe et coûteux, en particulier pour des modèles financiers sophistiqués ou lorsqu'un grand nombre de produits financiers doivent être calibrés simultanément.

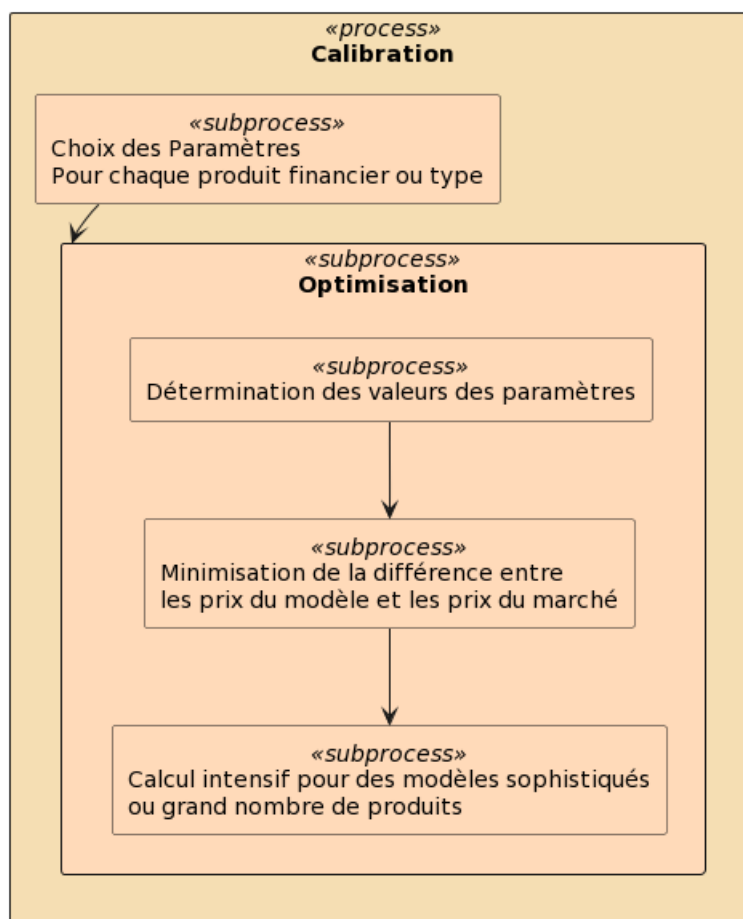


FIGURE 3.1.2 – Étapes de la calibration d'un trade

3.1.2.2 Stress Testing

Après la calibration des modèles aux données de marché actuelles, il est crucial de tester comment ces modèles se comporteraient dans des conditions de marché inhabituelles ou extrêmes. C'est ici qu'intervient le stress testing.

Le stress testing consiste à évaluer la stabilité d'un système financier en simulant des scénarios de crise. Le but est de déterminer dans quelle mesure les actifs financiers d'une institution ou d'un portefeuille pourraient être affectés par des changements radicaux et défavorables des conditions du marché.

Il s'agit d'un processus par lequel le trade est soumis à divers scénarios défavorables pour évaluer sa robustesse et sa résilience face à des conditions de marché exceptionnelles ou imprévues. Cela aide à comprendre les pires scénarios possibles et à se préparer en conséquence.

Généralement, les trades sont représentés sous forme d'un graphe où la structure est telle que la racine est le trade lui-même, et les nœuds feuilles sont les données de marché (market data) sur lesquelles ce trade repose. Cette représentation est particulièrement pertinente car elle visualise clairement les dépendances d'un trade par rapport aux divers facteurs du marché. Chaque feuille, représentant une donnée de marché spécifique, est un élément qui, lorsqu'il change, peut influencer la valeur ou le risque associé au trade.

Ainsi, lors de l'évaluation des risques d'un trade, un scénario est défini en effectuant des opérations sur une ou plusieurs de ces feuilles comme dans la figure suivante :

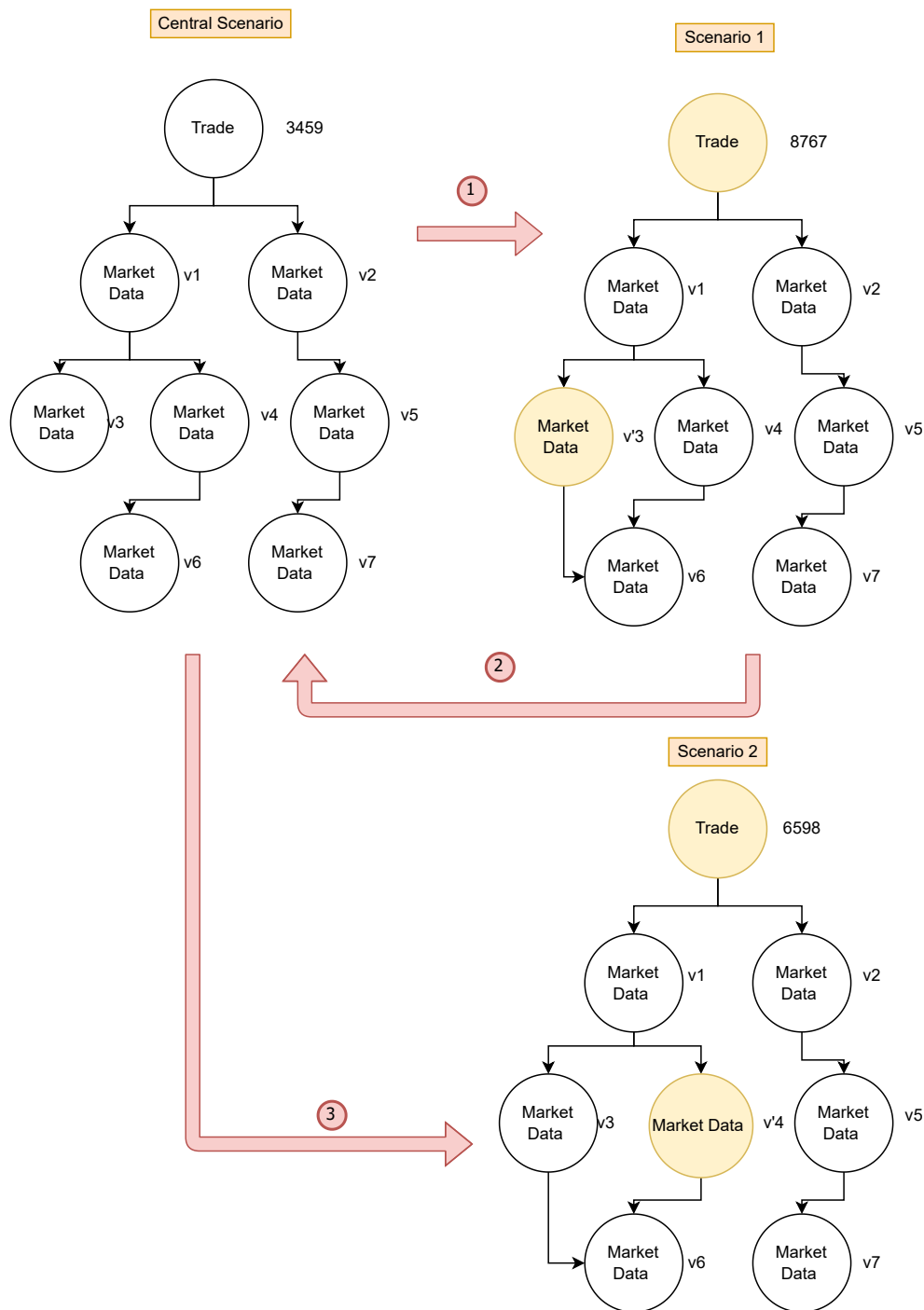


FIGURE 3.1.3 – Création de scénario d'un trade

3.1.3 Bibliothèques de pricing

Dans le contexte de notre problématique, il est nécessaire d'avoir recours à des outils spécialisés pour accomplir les tâches requises. Les bibliothèques de pricing, conçues pour le calcul et l'analyse de produits financiers, sont de ces outils indispensables. Il existe plusieurs bibliothèques de pricing sur le marché, chacune offrant ses propres avantages et inconvénients, et le choix d'une bibliothèque particulière dépendra fortement des besoins spécifiques du projet.

Parmi les bibliothèques disponibles, deux se sont distinguées lors de notre analyse : OpenGamma Strata et Open Source Risk Engine (ORE). Ces deux bibliothèques sont largement reconnues et utilisées dans le domaine de la finance de marché pour leurs fonctionnalités puissantes et leur fiabilité.

Toutefois, le choix de la bibliothèque à utiliser n'est pas une décision à prendre à la légère. Il est nécessaire de mener une étude approfondie des caractéristiques et des capacités de chaque bibliothèque pour déterminer celle qui sera la plus à même de répondre aux exigences du projet.

Dans la suite de cette section, nous allons explorer en détail chacune de ces bibliothèques, en comparant leurs fonctionnalités, leurs avantages et leurs inconvénients. Notre objectif est de justifier le choix que nous avons fait pour ce stage, tout en gardant à l'esprit notre objectif principal, qui est l'émulation de Murex.

Ainsi, notre évaluation se concentrera principalement sur les produits et trades couverts par chaque bibliothèque et sur la façon dont elles gèrent le flux de données.

3.1.3.1 Open Gamma Strata

Le premier mois du stage avait pour but de se lancer sur Opengamma Strata et reprendre un projet déjà faits par un stagiaire interne à Murex, qui va me permettre de comprendre comment cette bibliothèques marche , comment je pourrai enrichir le projets existants , comprendre la création des trades sur OpenGamma Strata et savoir faire les calculs basique du NPV(Net Present Value) sur une varieté des produits.

Strata [2], développée par OpenGamma, est une bibliothèque financière construite en Java. Elle se compose de plusieurs modules, chacun gérant différents aspects du calcul et de l'analyse financière. La bibliothèque est structurée de manière à être modulaire et flexible, offrant un éventail complet de fonctionnalités pour la tarification, le risque et la gestion de portefeuille.

La force de Strata réside dans son architecture modulaire, qui divise le système en différents composants qui peuvent être sélectionnés et utilisés en fonction des besoins spécifiques du projet. Chaque module est conçu pour traiter un aspect spécifique de l'analyse financière. Parmi les modules les plus notables, on retrouve :

- **Strata-Market** : Ce module est responsable du traitement des données de marché. Il fournit les interfaces et les implémentations pour gérer les différents types de données de marché, tels que les taux d'intérêt, les prix des actions, les taux de change, etc. Il permet également de gérer les ajustements de marché et les conventions de marché spécifiques à une région ou à un marché.
- **Strata-Measure** : Ce module est centré sur le calcul des mesures financières, comme la valeur actuelle, la sensibilité aux prix, les mesures de risque, etc. Il prend en entrée les données de marché et les détails des transactions, et produit les mesures financières requises.
- **Strata-Pricer** : Ce module fournit les algorithmes pour l'évaluation des instruments financiers. Il couvre une gamme d'instruments, y compris les swaps, les options, les futures, les obligations, etc.

- **Strata-Report** : Ce module s'occupe de la génération de rapports. Il peut créer des rapports détaillés sur les mesures financières calculées, y compris des analyses de risque et des analyses de performance.
- **Strata-Collect** : Ce module fournit une gamme d'outils pour la collecte, le nettoyage et la validation des données. Cela comprend la gestion des erreurs de données, le contrôle de la qualité des données, le filtrage des données, etc.

Produits Strata propose une couverture exhaustive des produits financiers. Voici une liste des produits couverts par Strata :

Produit	Strata
Taux d'intérêt	✓
Cap/Floor	✓
FX	✓
Volatilités	✓
Credit Default Swap	✓
Bond	✓

TABLE 3.1.1 – Comparaison des produits couverts par Strata

Trades en Strata En ce qui concerne la gestion des trades, Strata offre une approche bien structurée. Chaque produit financier a sa propre classe de trade, qui est construite en utilisant un builder pattern. Ce pattern assure que les objets sont créés de manière cohérente et correcte. Par exemple, pour un swap, Strata utilise une classe SwapTrade, qui est construite de la manière suivante :

```
SwapTrade swapTrade = SwapTrade.builder()
    .product(Swap.of(payLeg, receiveLeg))
    .info(TradeInfo.builder()
        .id(StandardId.of("example", "2"))
        .addAttribute(TradeAttributeType.DESCRPTION, "Libor 3m + spread vs Libor 6m")
        .counterparty(StandardId.of("example", "A"))
        .settlementDate(LocalDate.of(2014, 9, 12))
        .build())
    .build();
```

On peut aussi créer des trades à partir de fichiers FpML (Financial product Markup Language). FpML est un standard basé sur XML qui est largement utilisé dans le secteur financier pour communiquer des informations sur les produits dérivés et les transactions.

Pour importer un trade à partir d'un fichier FpML, vous pouvez utiliser la classe FpmlDocumentParser de Strata. Voici un exemple de la façon dont cela peut être fait :

```
File file = new File("path/to/your/file.fpml");
String fpml = Files.readString(file.toPath());
FpmlDocument document = FpmlDocumentParser.parse(fpml);
Trade trade = document.toTrade();
```

Voici une liste exhaustive des trades couverts sur Strata :

Trade
Swap
Cap/Floor
FX Spot
FX Forward
FX Volatility
Credit Default Swap
Bond
Bond Option
Bond Future
Bond Future Option
Swaption

TABLE 3.1.2 – Liste des trades couverts par Strata

MarketData en Strata La gestion des données de marché est un élément crucial pour les calculs financiers et la gestion des risques. Strata offre une classe spécifique pour cela, appelée `MarketData`. Elle fournit une approche unifiée pour gérer les diverses données de marché dont une application pourrait avoir besoin, notamment les courbes de taux d'intérêt, les surfaces de volatilité, les taux de change, et plus encore.

Un objet `MarketData` est immuable, ce qui signifie qu'une fois créé, il ne peut pas être modifié. Cela garantit que les données de marché restent cohérentes tout au long de leur utilisation. Si vous avez besoin de modifier les données de marché, vous devez créer un nouvel objet `MarketData`.

La création d'un objet `MarketData` est facilitée par l'utilisation d'une classe builder, comme dans l'exemple suivant :

```
MarketData marketData = ImmutableMarketData.builder(date)
.addValue(CurveId.of("Default", "USD-Disc"), discountCurve)
.addValue(CurveId.of("Default", "USD-3ML"), iborCurve)
.build();
```

Retour d'expérience sur Strata Toutefois, bien que Strata soit une bibliothèque puissante et flexible, elle présente certaines limitations qui doivent être prises en compte. Par exemple, bien qu'elle offre une large gamme de fonctionnalités, toutes ne sont pas nécessairement pertinentes ou utiles pour tous les projets. De plus, la mise en place de Strata peut nécessiter une certaine expertise technique et une compréhension approfondie de la bibliothèque, ce qui peut s'avérer être un défi même pour des utilisateurs expérimentés. On signale aussi que `OpenGamma` sont des concurrents directs de Murex et toutes demande d'aide sera risquée.

3.1.3.2 Open Source Risk Engine

La deuxième bibliothèque explorée au cours de ce stage est l'Open Source Risk Engine (ORE)[3]. Contrairement à Strata, ORE a démontré une couverture de produits plus exhaustive et une compatibilité accrue avec les exigences du projet. D'autre part, ORE bénéficie d'un support solide de la communauté, avec des mises à jour régulières et l'ajout de nouvelles fonctionnalités.

ORE, initialement développé par Quaternion Risk Management en collaboration avec l'Institute for Advanced Studies de l'Université de Heidelberg, est une bibliothèque financière open source développée en C++ disponible en libre accès via ce lien. Cette bibliothèque met l'accent sur les produits de crédit et de taux d'intérêt, et intègre une suite de modèles de risque de crédit et de marché.

Enrichi par la bibliothèque de calculs financiers QuantLib, ORE étend ses capacités en ajoutant des fonctionnalités pour la modélisation du risque de crédit, la gestion des données de marché et la production de rapports réglementaires. De plus, ORE offre une API Python, permettant ainsi de construire des applications et scripts personnalisés.

Dans le contexte de notre étude, une comparaison approfondie entre Strata et ORE est essentielle. Cette comparaison se concentrera sur des critères clés tels que la couverture des produits, l'architecture, la gestion des données de marché, et la facilité d'utilisation. Cela nous aidera à déterminer quelle bibliothèque serait la plus appropriée pour émuler MX.

Dans les sections suivantes, nous détaillerons les aspects importants de l'ORE, et comment il peut être utilisé dans le cadre de notre objectif d'émuler MX.

Flux de Données Les étapes principales de traitement suivies par ORE pour produire des résultats d'analyse de risque sont schématisées dans la Figure 3.1.4. Tous les calculs et résultats de ORE sont générés en trois étapes fondamentales comme indiqué dans les trois boîtes dans la partie supérieure de la figure. À chaque étape, des données appropriées (décrites ci-dessous) sont chargées et les résultats sont générés, soit sous forme de rapport lisible par l'homme, soit dans une étape intermédiaire comme des fichiers de données pures (par exemple, données de NPV).

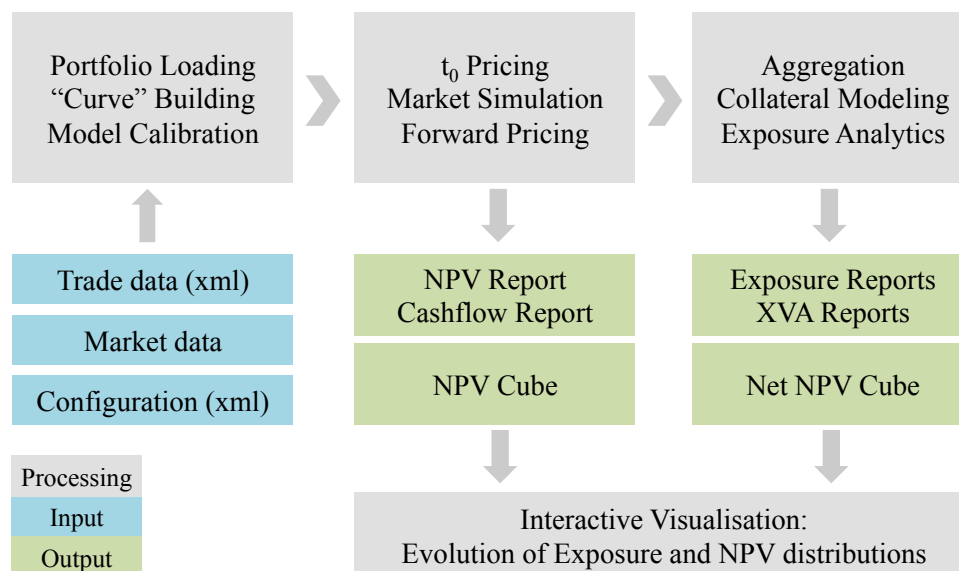


FIGURE 3.1.4 – Esquisse du processus ORE, des entrées et des sorties.

Le processus général d'ORE doit être paramétré à l'aide d'un ensemble de fichiers de configuration XML. Les données de marché sont fournies dans un simple fichier texte à trois colonnes, fichiers txt ou csv.

La première étape de traitement (boîte en haut à gauche) comprend le chargement du portefeuille à analyser, la construction des courbes de rendement nécessaires à la tarification et l'étalonnage des modèles de tarification et de simulation.

La deuxième étape de traitement (boîte du milieu en haut) est l'évaluation du portefeuille, la génération de flux de trésorerie, l'analyse de risque conventionnelle telle que l'analyse de sensibilité et les tests de stress, et enfin, la simulation du marché et l'évaluation du portefeuille dans le temps sous les scénarios de Monte Carlo. Cette étape produit un **cube NPV**, c'est-à-dire des NPVs par trade, scénario et future date d'évaluation.

Il est à noter que nous nous intéressons particulièrement à la calibration, au stress testing et à l'analyse NPV.

Trades dans ORE Dans ORE, les trades sont gérés d'une manière légèrement différente de celle de Strata. Au lieu d'utiliser une structure de builder comme Strata, ORE utilise des fichiers XML pour définir et gérer les trades. Chaque trade est défini dans un fichier XML séparé qui comprend toutes les informations nécessaires pour définir le trade, y compris les détails sur les instruments financiers utilisés, les dates de transaction, et toute autre information pertinente. Ceci va dans le but du stage car Mx utilise des formats de fichiers similaires à celle de ORE, qui s'appellent MxML. Les trades sont aussi représentés en XML versionné de Murex. Des exemples de représentation XML des trades sur ORE est disponible via ce lien [5].

En termes de gestion des trades, ORE se distingue également par sa souplesse et sa diversité de couverture. Comme observé dans le tableau ci-dessus, une variété de produits financiers peuvent être gérés dans ORE, y compris des options sur matières premières, des swaps, des produits à taux fixe, des produits dérivés de crédit, des produits dérivés d'équité et d'autres produits financiers plus complexes. Cette diversité de couverture signifie que ORE peut être utilisé pour analyser et gérer les risques associés à un large éventail de portefeuilles financiers.

Trade Type in ORE
CommodityForward
CommodityOption
Bond
BondOption
BondRepo
BondTRS
IndexCreditDefaultSwap
IndexCreditDefaultSwapOption
EquityForward
EquityOption
EquitySwap
Swap
FxForward
FxOption
FxSwap
CapFloor
ForwardRateAgreement
Swaption

TABLE 3.1.3 – Types de trades pris en charge par ORE

3.1.4 Prise en main d'ORE

Lors de la phase d'apprentissage de l'outil ORE, l'objectif était de manipuler des trades, exécuter des tests de calcul, et comprendre en profondeur le comportement d'ORE. Plus particulièrement, l'observation se focalisait sur le temps d'exécution pour chaque type de trade sur les aspects suivants :

- Construction des markets data
- Construction du portfolio
- Calcul de la valeur actuelle nette ou NPV

Des défis ont été rencontrés lors de cette exploration. Ces derniers étaient surtout liés à la compréhension de la structure des fichiers XML qui sont nécessaires pour le fonctionnement d'ORE, le débogage des erreurs de calcul, et la gestion de la performance.

3.1.4.1 Première expérience

La première étape de cette expérience a consisté à créer des trades pour chacun des types de produits mentionnés dans le tableau 3.1.3.

Un fichier XML a été créé pour chaque type de trade. Ce fichier comportent les informations sur le trade créée comme expliqué dans le guide de ORE [4].

Afin de simplifier le processus de calcul, tous les trades créés utilisent les mêmes conventions, paramètre de données de marchés, qui sont aussi des fichiers xml comme expliqués dans la partie précédente.

Le binaire de ORE prend comme un paramètre un fichier XML, (ore.xml) qui contient alors les chemins du portfolio, les données du marchés ainsi que le type de calculations souhaités comme suit :

Listing 3.1 – ore.xml file for Swap NPV label

```

1 <?xml version="1.0"?>
2 <ORE>
3   <Setup>
4     <Parameter name="asofDate">2016-02-05</Parameter>
5     <Parameter name="inputPath">Input</Parameter>
6     <Parameter name="outputPath">Output</Parameter>
7     <Parameter name="logFile">log.txt</Parameter>
8     <Parameter name="logMask">31</Parameter>
9     <Parameter name="marketDataFile">../../Input/market_20160205_flat.txt</Parameter>
10    <Parameter name="fixingDataFile">../../Input/fixings_20160205.txt</Parameter>
11    <Parameter name="implyTodaysFixings">Y</Parameter>
12    <Parameter name="curveConfigFile">../../Input/curveconfig.xml</Parameter>
13    <Parameter name="conventionsFile">../../Input/conventions.xml</Parameter>
14    <Parameter name="marketConfigFile">../../Input/todaysmarket.xml</Parameter>
15    <Parameter name="pricingEnginesFile">../../Input/pricingengine.xml</Parameter>
16    <Parameter name="portfolioFile">portfolio_swap.xml</Parameter>
17    <Parameter name="observationModel">Disable</Parameter>
18  </Setup>
19  <Markets>
20    <Parameter name="lgmcalibration">libor</Parameter>
21    <Parameter name="fxcalibration">libor</Parameter>
22    <Parameter name="pricing">libor</Parameter>
23    <Parameter name="simulation">libor</Parameter>
24  </Markets>
25  <Analytics>
26    <Analytic type="npv">
27      <Parameter name="active">Y</Parameter>
28      <Parameter name="baseCurrency">EUR</Parameter>
29      <Parameter name="outputFileName">npv.csv</Parameter>
30    </Analytic>
31    <Analytic type="cashflow">
32      <Parameter name="active">N</Parameter>
33      <Parameter name="outputFileName">flows.csv</Parameter>
34    </Analytic>
35    <Analytic type="xva">
36      <Parameter name="active">N</Parameter>
37    </Analytic>
38  </Analytics>
39 </ORE>

```

Un calcul simple et basique a été faits sur chaque type de trade. C'est le NPV.

Le tableau suivant montre le temps d'exécution pour chaque type de trade. Les valeurs ont été récupérées des logs créés par ORE.

Type de Transaction	Temps d'Exécution en S
ForwardRateAgreement	4.264285
BondRepo	4.917384
BondTRS	4.988241
EquityOption	4.932696
Swap	4.826509
EquityForward	5.044586
Swaption	5.196813
EquitySwap	5.845465
Bond	6.545547
FxForward	6.410433
FxOption	6.437908
FxSwap	6.442984
IndexCreditDefaultSwap	6.704794
BondOption	7.914845
CommodityForward	8.484245
CommodityOption	10.421262
CapFloor	21.4139113333

TABLE 3.1.4 – Valeurs du temps d'exécution pour différents types de trades

En observant ce tableau, on remarque certaines tendances et des variations significatives dans les temps d'exécution selon les types de trades. Voyons en détail :

- A. **Faible temps d'exécution pour 'FRA' :** L'opération la plus rapide à s'exécuter est 'ForwardRateAgreement', avec un temps de seulement 4.264285 unités. C'est logique car un FRA est un trade simple avec peu d'informations contrairement aux autres trades. Il suffit de regarder le XML dans le guide de ORE.
- B. **Temps d'exécution élevé pour 'CapFloor' :** À l'opposé, 'CapFloor' a le temps d'exécution le plus élevé à 21.4139113333 unités. Ceci nous a emmené à ce posés des question sur ORE ; Pourquoi le temps d'exécution est assez élevés comparèrent aux autres trades ?
- C. **Transactions 'Fx' :** Les transactions telles que 'FxForward', 'FxOption' et 'FxSwap' ont des temps d'exécution très similaires, se situant tous autour de 6.4 unités ce qui est cohérent car elles partagent une logique de traitement similaire sur ORE.
- D. **Transactions 'Bond' :** Si nous regardons les transactions liées aux obligations comme 'Bond', 'BondRepo', 'BondTRS' et 'BondOption', nous pouvons observer que bien que leurs natures soient différentes, leurs temps d'exécution se situent dans une plage relativement étroite, allant de 4.9 à 7.9 unités.

Conclusion de la première expérience La dispersion des temps d'exécution varie significativement, allant de moins de 1 à plus de 21 unités sur les trades sur ORE. Pourtant , il faut pousser la recherches des temps d'exécution de trades et ajouter plus de logs.

3.1.4.2 Deuxième expérience

L'intérêt de la deuxième expérience est de vérifier la validité des résultats de la première expérience.

Pour cela, on utilise l'outil de profiling de Visual studio pour regarder l'enchaînement d'appels de fonction sur ORE.

On lance le profiler sur tous les exemples de trades déjà créés. Par exemple pour l'exemple du swap, en lançant le profiler, on a le fichier SwapReport.diagsession suivant :

ExpectedOutput	25/05/2023 12:13	File folder	
Input	25/05/2023 11:35	File folder	
Output	25/05/2023 10:37	File folder	
ore.log	05/06/2023 15:18	Text Document	1 KB
run.py	25/05/2023 11:19	Python Source File	1 KB
<u>SwapReport.diagsession</u>	26/05/2023 16:42	Diagnostic Sessio...	4 112 KB

FIGURE 3.1.5 – Fichier Rapport de l'exemple de swap

En ouvrant le fichier précédant, on a alors la figure suivante 3.1.8 qui montre la pile d'appel de ORE :

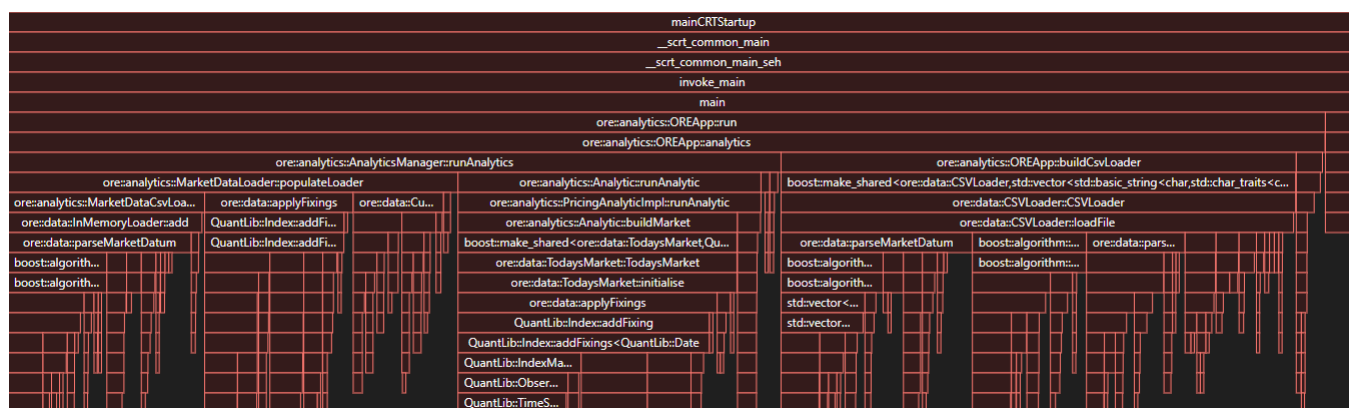


FIGURE 3.1.6 – Pile d'appel de ORE pour calcul d'un NPV sur un Swap

On zoome alors sur la fonction ore : :analytics : :Analytics : :runAnalytic comme dans la figure suivante :

ore::analytics::OREApp::run	6850 (95,02 %)	0 (0,00 %)	ore
ore::analytics::OREApp::analytics	6849 (95,01 %)	0 (0,00 %)	ore
ore::analytics::AnalyticsManager::runAnalytics	4846 (67,22 %)	0 (0,00 %)	ore
ore::analytics::Analytic::runAnalytic	3000 (41,61 %)	0 (0,00 %)	ore
ore::analytics::PricingAnalyticImpl::runAnalytic	3000 (41,61 %)	0 (0,00 %)	ore
ore::analytics::Analytic::buildPortfolio	1906 (26,44 %)	0 (0,00 %)	ore
ore::analytics::Analytic::buildMarket	1086 (15,06 %)	0 (0,00 %)	ore
ore::analytics::ReportWriter::writeNpv	8 (0,11 %)	0 (0,00 %)	ore

FIGURE 3.1.7 – Pile d'appel de runAnalytic pour calcul d'un NPV sur un Swap

Les deux principales fonctions appelées sont alors :

MarketBuild : C'est la fonction qui construit les données du marché. Elle appelle deux autres fonction. La première étant TodaysMarket qui construit les données du marchés du jour et build-DependencyGraph qui construit le graphe des markets data et les lie au trades correspondants.

BuildPortfolio : C'est la fonction qui construit les objets trades en lisant leurs fichiers XML.

On ajoute alors des logs supplémentaire dans ces fonctions pour calculer leurs vrais temps d'exécution dans le code de ORE. La librairie Log4cxx[12] a été utilisé pour les logs. Un fichier ore.log est crée et contient alors le temps d'appel , comme dans la figure suivante :

```
Engine\OREData\ored\marketdata\todaysmarket.cpp::TodaysMarket:181]: Time to build the dependency graphs : 0.055154 seconds  
Engine\OREAnalytics\orea\app\analytics\pricinganalytic.cpp::Pricing Analytics:91]: Time to build market: 1.10421 seconds  
Engine\OREAnalytics\orea\app\analytics\pricinganalytic.cpp::Pricing Analytics:99]: Time to build portfolio: 1.91302 seconds  
Engine\OREAnalytics\orea\app\analyticsmanager.cpp::Run Analytics:194]: Time to run analytics: 4.89246 seconds  
Engine\App\ore.cpp::ORE App:102]: Time to run the request: 7.27141 seconds
```

FIGURE 3.1.8 – Fichiers orelog.txt sur l'exemple du Swap

En faisant la même opération sur tous les autres types de trades, on a le graphe suivant :



FIGURE 3.1.9 – Temps d'exécution en s pour différents trades sur ORE

En observant ce graphe , nous pouvons formuler les remarques suivantes :

- Le temps d'exécution "RunTime" pour les différents types de trades montre des variations notables. CapFloor présente le temps d'exécution le plus long à 10.078 secondes, indiquant une possible complexité ou un volume de données élevé pour ce type de trade. En revanche, FRA a le temps d'exécution le plus court à seulement 5.103 secondes. La plupart des autres trades ont des temps d'exécution se situant entre 5 et 8 secondes, suggérant une certaine uniformité pour ces instruments. Ceci est toujours cohérents avec les résultats de la première expérience.
- En moyenne , le market build est aussi presque le même. Ceci est logique car on utilise les mêmes données du marchés. De même pour la construction du graphe de dépendance et todays Market, car comme précisé précédemment, ils sont appelé à l'intérieur du market Build.

- Dans la fonction Build Portfolio , on remarque une grosse différence entre les valeurs. CapFloor prend le plus de temps à exécuter avec une durée de 4.838 secondes. Les trades de types Option ,BondOption et Swaption suivent respectivement avec des durées d'exécution de 3.195 et 2.420 secondes. BondRepo présente le temps d'exécution le plus court à seulement 0.086 secondes. La plupart des autres types de trades, tels que EquityOption, EquitySwap, CommodityOption, et ICDS, ont des temps d'exécution inférieurs à 0.5 secondes.

On peut alors classer les trades sur ORE selon le temps nécessaire à leur construction avec la fonction "BuildPortfolio". Cette classification nous mène à trois catégories distinctes : "Grand", "Moyen" et "Petit", représentant respectivement les trades ayant des temps d'exécution longs, moyens et courts :

Type de Trade	Temps d'Exécution	Classe
CapFloor	4.838	Grand
BondOption	3.195	Grand
Swaption	2.420	Grand
Swap	1.906	Grand
FxForward	1.893	Moyen
FxSwap	1.888	Moyen
FxOption	1.869	Moyen
FRA	1.020	Moyen
EquityOption	0.224	Moyen
EquitySwap	0.221	Moyen
CommodityOption	0.215	Moyen
ICDS	0.211	Moyen
CommodityForward	0.204	Moyen
EquityForward	0.200	Petit
Bond	0.170	Petit
BondTRS	0.168	Petit
BondRepo	0.086	Petit

TABLE 3.1.5 – Classification des types de trades selon le temps d'exécution de BuildPortfolio

Conclusion La deuxième expérience confirme les résultats de la première et pousse en plus en zoomant sur le critère qui différencie le temps d'exécution sur ORE pour un calcul simple de NPV. Une troisième étude sera faite après la partie engin de calcul pour vérifier la variabilités sur des lots de trades du même type et en ajoutant le stress test.

3.1.5 Engin de calcul

Dans le but de simuler Mx, un engin de calcul sera fait en se basant sur ORE. En effet, ORE sera servi comme base de calculateur et d'autres composantes seront créées afin d'élargir les capacités de ORE sur un large portefeuille.

3.1.5.1 Architecture

La figure suivante 3.1.10 représente l'architecture de l'engin en mode séquentiel

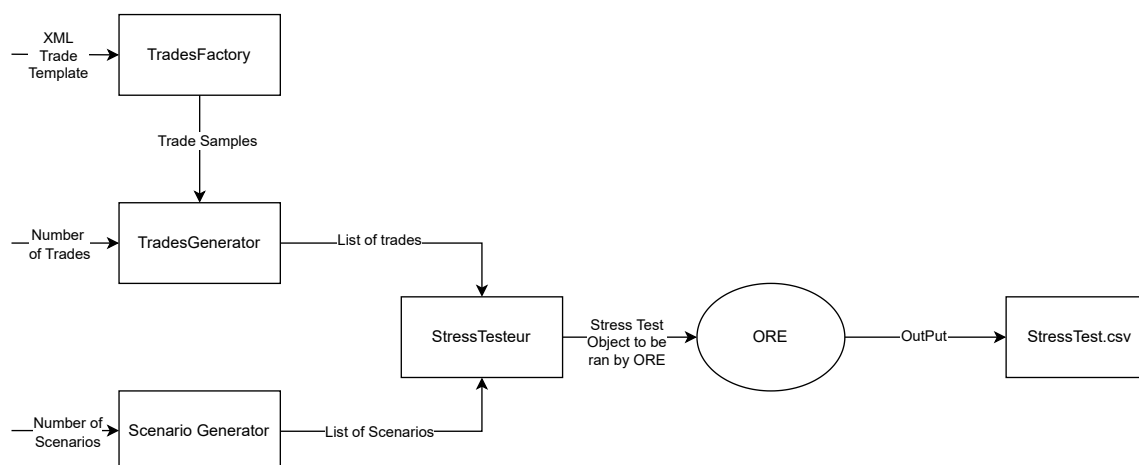


FIGURE 3.1.10 – Architecture de l'engin

TradesFactory : Cette classe prend en entrée des fichiers XML qui représentent les templates des trades en XML.

TradesGenerator : On génère un nombre N de trades selon une stratégie à définir.

ScenarioGenerator : On génère un nombre N de scénarios.

StressTester : Cette classe prend en entrée les trades et les scénarios et les données du marché qui sont statiques dans notre cas. On retourne un objet stress test que ORE exécute.

ORE : prend en entrée un objet stress test à calculer et retourne un fichier stresstest.csv.

3.1.5.2 Générateur de trades

Le générateur de trades ou "Trades Generator" est conçu pour faciliter la création automatisée de différents types de trades basés sur des modèles XML. Cette automatisation permet de gérer efficacement un grand volume de trades sans avoir à les saisir manuellement, ce qui est crucial pour les simulations de marché à grande échelle pour tester la robustesse d'un système.

La classe TradesFactory joue le rôle d'une usine de fabrication de trades. Elle offre des méthodes pour convertir un fichier XML en une chaîne de caractères et pour transformer un objet Trade en sa représentation XML. La méthode clé, createTradeFromXML, initialise et construit un trade à partir d'un fichier XML donné.

La classe TradeGenerator prend le relais en utilisant TradesFactory pour générer des trades basés sur des modèles XML fournis. Plusieurs modes de génération sont proposés :

- La génération de trades de type spécifique, où tous les trades créés sont du même type.
- La génération aléatoire, où les types de trades sont choisis au hasard.
- Une distribution égale (round robin) de tous les types de trades disponibles.
- Une génération basée sur des pourcentages prédéfinis, illustrée par le mode "jad", où chaque type de trade est généré selon un pourcentage spécifique du nombre total de trades. Jad travaille à Murex dans le département MSA Instruments et nous a fournis les pourcentage des trades et produits utilisés par les clients de Murex, qui sont données dans le tableau suivant 3.1.6 :

Type de Trade	Pourcentage
Bond	20.9
BondRepo	0.9524
BondTRS	1.5238
CommodityOption	7.619
CommodityForward	3.8095
CapFloor	0.2857
EquityForward	0.3809
EquityOption	1.9048
EquitySwap	23.0857
FxOption	5.7143
FxForward	7.619
FxSwap	6.3905
IndexCreditDefaultSwap	4.7619
Swap	1.619
Swaption	4.8667
ForwardRateAgreement	3.8095

TABLE 3.1.6 – Pourcentages de la configuration "Jad"

3.1.5.3 Générateur de scénarios

Le générateur de scénarios, implémenté via la méthode `generateStressScenario`, est le cœur de la classe. Cette méthode prend en entrée un identifiant, souvent représentatif d'un ensemble d'instruments financiers ou d'une condition de marché spécifique, et génère un scénario de stress adapté.

L'algorithme commence par identifier l'ensemble de données pertinent pour cet identifiant. Par la suite, des courbes spécifiques sont générées, telles que :

- Les courbes de remise (Discount Curves)
- Les courbes d'index
- Les courbes de rendement
- Les variations des spots FX

Chaque courbe est ajustée selon des critères définis pour modéliser les conditions de stress. Ces critères peuvent être basés sur des événements historiques, des prédictions de marché ou des hypothèses de risque.

Enfin, le scénario de stress est retourné, prêt à être appliqué sur les instruments financiers ou portefeuilles concernés pour évaluation.

3.1.5.4 Stress Tester

La classe **StressTester** se compose essentiellement de la méthode **performStressTest**. Elle fournit une procédure complète pour configurer et exécuter un test de stress sur un portefeuille financier. Elle utilise plusieurs classes et objets pour définir les paramètres du marché, générer les scénarios de stress et construire les composants nécessaires pour effectuer le test de stress.

Étape 1 : Configuration initiale La date d'évaluation est fixée au 14 avril 2016. Une série de devises est ensuite définie, avec l'EUR comme devise de base, suivie de GBP, CHF, USD et JPY.

Étape 2 : Configuration du marché de stress Un objet **StressMarket** est créé, à partir duquel un ensemble de paramètres pour le marché simulé est configuré via la méthode **setupStressSimMarketData**.

Étape 3 : Configuration du scénario de stress Un objet **StressScenario** est créé, et le nombre spécifié de scénarios de stress est généré à l'aide de la méthode **generateStressScenarios**.

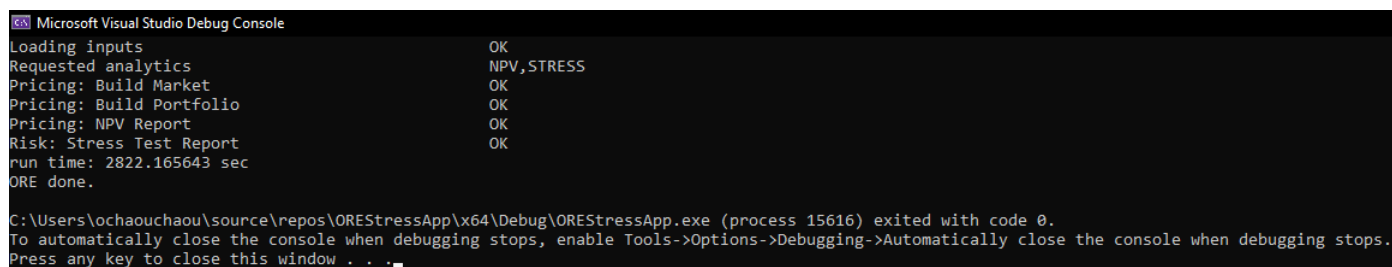
Étape 4 : Mise en place du marché scénario L'objet **StressMarket** est utilisé pour configurer le marché de simulation. Un marché de simulation est ensuite créé en utilisant les paramètres du marché simulé définis précédemment et le marché initial donné comme argument à la méthode.

Étape 5 : Configuration du portefeuille Le portefeuille est généré à partir des trades fournies par **TradesGenerator**.

Étape 6 : Construction de l'objet d'analyse de sensibilité Finalement, un objet de **StressTest** est construit en utilisant le portefeuille, le marché initial, la configuration par défaut, les données du moteur, les paramètres du marché simulé et les données de scénario de stress.

3.1.5.5 Résultats du calcul séquentiel

Premier Test : On lance l'engin du calcul sur 1000 trades suivant la configuration "Jad" et 500 scénarios. Le temps d'exécution est 47 min, voir la figure suivante 3.1.11 :



```
Microsoft Visual Studio Debug Console
Loading inputs OK
Requested analytics NPV, STRESS
Pricing: Build Market OK
Pricing: Build Portfolio OK
Pricing: NPV Report OK
Risk: Stress Test Report OK
run time: 2822.165643 sec
ORE done.

C:\Users\ochaouchaou\source\repos\OREStressApp\x64\Debug\OREStressApp.exe (process 15616) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . . _
```

FIGURE 3.1.11 – Résultats de calcul pour 1000 Trades x 500 Scénarios

Commentaires premier test : Le temps de calcul a augmenté, ce qui est normal, car la charge a aussi augmenté.

Deuxième Test : On lance de façon parallèle 5 threads. Chaque thread lance l'engin de la façon suivante :

Thread	Number of trades	Type de trade	Number of Scenarios	Runtime (min.)
Thread 1	200	Bond	500	3.001
Thread 2	200	EquityForward	500	3.191
Thread 3	200	FxForward	500	3.494
Thread 4	200	Swap	500	5.991
Thread 5	200	Swaption	500	85.927

TABLE 3.1.7 – Temps d'exécution pour chaque thread

Commentaires deuxième test : Le temps d'exécution change selon le type de trade donné, même en montée de charge, ce qui valide les premières expériences.

Question :

Comment pouvons-nous optimiser les performances de notre engin face à une augmentation conséquente du nombre de trades et de scénarios ?

3.2 Application de la technologie du cache au service

Suite à notre exploration des défis liés à l'amélioration des performances de notre moteur, en particulier lors de montées en charge avec un nombre croissant de trades et de scénarios, il est devenu impératif de chercher des solutions innovantes. L'adoption d'une technologie de cache s'est imposée comme une réponse logique à ces défis. Le caching permet non seulement de réduire la latence et d'améliorer la réactivité d'une application, mais aussi de gérer efficacement les montées en charge. Dans cette quête d'optimisation, nous nous sommes tournés vers une solution éprouvée et largement adoptée : Redis.

Avec la croissance exponentielle des données et la demande pour des temps de réponse rapides, l'optimisation des performances des services est devenue une priorité pour de nombreuses entreprises. L'un des moyens les plus efficaces d'atteindre une haute performance est l'utilisation de la technologie de cache. Le caching permet de stocker temporairement des données fréquemment utilisées dans une mémoire à accès rapide, réduisant ainsi la nécessité de récupérer ces données à partir d'une source plus lente, comme une base de données. Dans ce contexte, de nombreux outils et solutions de caching ont été développés, et parmi eux, Redis se distingue comme une option populaire et efficace.

3.2.1 Redis

Redis est une base de données en mémoire, utilisée comme serveur de cache, de courtage de messages et de file d'attente. Elle est renommée pour sa vitesse, sa fiabilité et sa polyvalence dans le traitement des structures de données. En intégrant Redis dans notre service, nous espérons améliorer considérablement les temps de réponse et réduire la charge sur les systèmes sous-jacents, tout en gérant efficacement les montées en charge.

3.2.1.1 Origines et évolution historique de Redis

Redis, acronyme pour "Remote Dictionary Server", a vu le jour en 2009 sous la houlette de Salvatore Sanfilippo. Initialement conçu pour pallier les problèmes de scalabilité rencontrés dans un projet de startup de Sanfilippo, Redis s'est rapidement distingué par sa simplicité et sa rapidité, gagnant une place prépondérante parmi les bases de données en mémoire. Conçu comme un magasin de structure de données, sa capacité à prendre en charge de multiples types de structures de données, des chaînes aux listes, en passant par les ensembles et les hachages, a renforcé son attrait au sein de la communauté de développeurs. Grâce à l'apport constant d'une communauté active et engagée, Redis a vu l'ajout de nombreuses fonctionnalités tout en préservant ses principaux atouts. Aujourd'hui, Redis est non seulement reconnu pour sa fonctionnalité de mise en cache mais sert aussi dans des scénarios diversifiés comme les courtiers de messages, et est adopté par de nombreuses grandes entreprises à travers le monde. Son évolution témoigne de sa capacité à s'adapter et à répondre aux exigences changeantes de l'industrie technologique.

3.2.1.2 Fonctionnement interne de Redis

Redis, au-delà de sa popularité, se distingue par la sophistication de son architecture interne. Pour comprendre pleinement pourquoi et comment Redis offre des performances optimales et se montre si flexible dans divers scénarios d'utilisation, il est essentiel d'examiner de près son fonctionnement interne. Cela passe notamment par la compréhension du mode de stockage qu'il privilégie, des structures de données qu'il prend en charge et des mécanismes qu'il met en œuvre pour garantir la persistance des données.

Principe du stockage en mémoire et ses avantages en termes de performance :

Contrairement aux bases de données traditionnelles qui stockent des données sur disque, Redis utilise la mémoire principale de l'ordinateur pour conserver l'ensemble de ses données. Ce choix architectural n'est pas anodin et constitue le socle des performances exceptionnelles qu'offre Redis. Le fait d'accéder aux données directement en mémoire permet de réduire drastiquement les temps de latence par rapport à une lecture sur un disque.

En termes plus techniques, la mémoire RAM offre des temps d'accès constants, souvent mesurés en nanosecondes, tandis que les disques, même les plus rapides (comme les SSDs), présentent des temps d'accès variables, généralement en microsecondes ou millisecondes. Ce gain en rapidité est d'autant plus sensible lorsque l'on traite un grand volume de requêtes en parallèle ou des opérations nécessitant des accès fréquents aux données.

Cependant, si le stockage en mémoire offre une vitesse incomparable, il présente aussi des défis, notamment en matière de volatilité : les données en RAM sont perdues lors d'un redémarrage ou d'une panne. Redis a toutefois prévu des mécanismes pour garantir la persistance des données, assurant ainsi une sécurité et une durabilité adaptées aux besoins des entreprises modernes.

Structures de données prises en charge :

L'un des aspects les plus distinctifs de Redis réside dans sa riche collection de structures de données. Alors que de nombreuses bases de données clés-valeur se limitent à des paires clé-valeur simples, Redis va bien au-delà, offrant une panoplie de structures qui permettent de modéliser de manière efficiente une grande variété de cas d'utilisation.

- *Chaînes (Strings)* : Il s'agit de la structure de données la plus simple de Redis, pouvant stocker une chaîne de caractères ou une valeur numérique. C'est également la base pour les opérations atomiques complexes telles que l'incrément.
- *Listes (Lists)* : Ce sont des collections ordonnées de chaînes, implémentées comme des listes liées. Elles sont particulièrement utiles pour implémenter des files d'attente, des piles et d'autres structures séquentielles.
- *Ensembles (Sets)* : Comme leur nom l'indique, ils permettent de stocker une collection non ordonnée d'éléments uniques. Les opérations classiques d'union, d'intersection ou de différence entre ensembles sont efficacement gérées par Redis.
- *Hachages (Hashes)* : Ils sont utilisés pour représenter des objets en associant un ensemble de champs à leurs valeurs respectives. C'est en quelque sorte une table clé-valeur imbriquée au sein d'une clé Redis.
- *Ensembles ordonnés (Sorted Sets)* : Ces structures permettent de maintenir une collection ordonnée d'éléments. Chaque élément est associé à un score, déterminant l'ordre dans l'ensemble.
- *Streams [6]* : Les streams Redis sont un type de structure de données qui peuvent être utilisées pour implémenter des journaux d'événements ou des files d'attente de messages. Ils supportent plusieurs producteurs et consommateurs et peuvent retenir de grandes quantités de messages sans perte de performance.

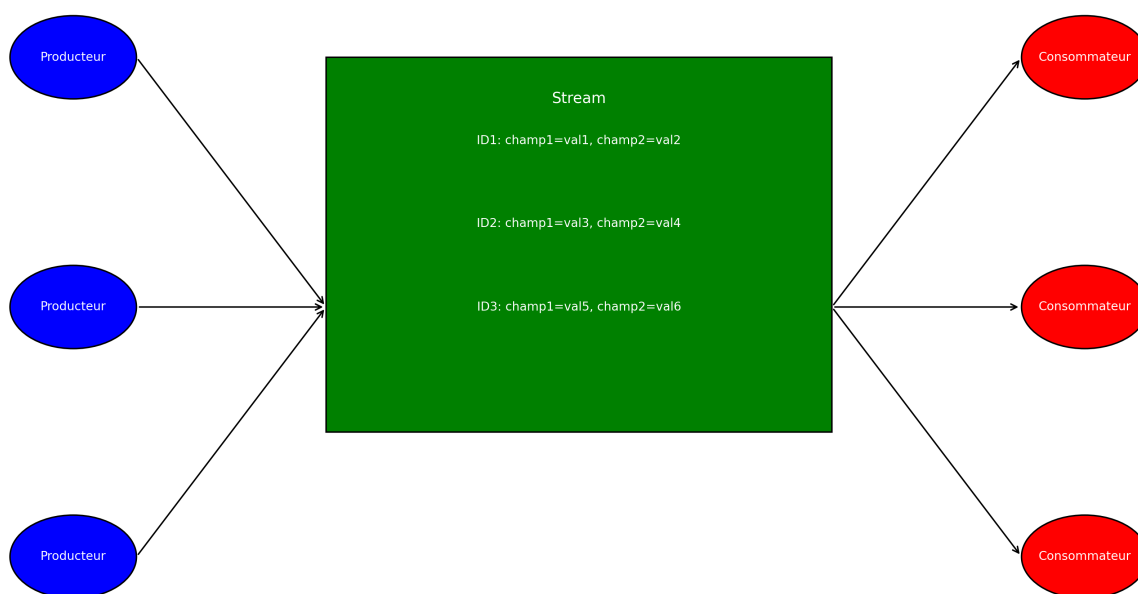


FIGURE 3.2.1 – Schéma illustrant le fonctionnement des streams Redis

- *Pub/Sub* [7] : Redis prend également en charge le paradigme de publication/abonnement. Cela permet à un nombre quelconque d'éditeurs de publier des messages vers un canal, et à un nombre quelconque d'abonnés de recevoir ces messages.

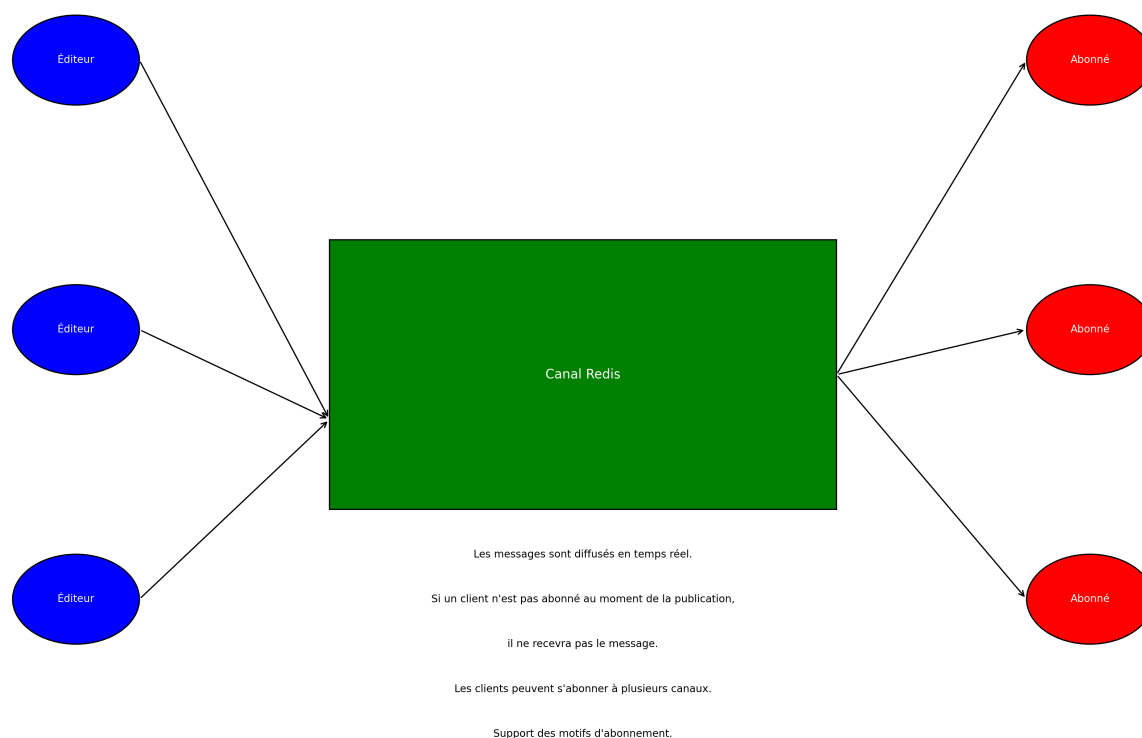


FIGURE 3.2.2 – Schéma illustrant le modèle Pub/Sub de Redis

- *Bitmaps, HyperLogLogs, GeoHashes, etc.* : Redis fournit également d'autres structures spécialisées qui permettent d'optimiser le stockage et le traitement pour des cas d'utilisation spécifiques.

Dans l'intégration de Redis avec notre engin, on utilisera les listes, les tables de hachage, les streams et le Pub/Sub.

Mécanismes assurant la persistance des données :

Redis, bien que principalement une base de données en mémoire, offre plusieurs mécanismes pour garantir la persistance des données, permettant ainsi de prévenir les pertes en cas de défaillances :

- **RDB (Redis Database File)** : À intervalles configurables, Redis crée une image à un moment donné de l'ensemble de ses données. Ces instantanés sont stockés sous forme de fichiers binaires sur le disque. Ils offrent une manière efficace de réaliser des sauvegardes complètes du jeu de données.
- **AOF (Append Only File)** : Avec cette méthode, toutes les opérations (ou commandes) qui modifient l'ensemble de données sont ajoutées à un fichier journal. Lors d'un redémarrage, Redis relit ce fichier pour reconstruire l'état complet de la base de données. L'AOF permet une durabilité fine, car il est possible de configurer la fréquence d'écriture des commandes sur le disque.

- **Persistance hybride** : Redis permet aussi d'utiliser conjointement RDB et AOF. Dans ce mode, il bénéficie de la rapidité des instantanés RDB tout en garantissant la sécurité des données grâce à la journalisation de l'AOF.

3.2.2 Mise en œuvre de Redis dans notre système - Première tentative

Notre solution repose sur une architecture robuste qui capitalise sur les avantages de Redis pour obtenir des performances optimales et une flexibilité accrue. Cette architecture, conçue pour fonctionner en mode distribué, garantit que chaque composant interagit harmonieusement, assurant ainsi une mise en œuvre efficace de notre système.

3.2.2.1 Vue d'ensemble de la 1^{ère} Architecture

La figure .0.1 offre une vue d'ensemble de cette architecture. Les différents éléments de notre architecture sont clairement illustrés, montrant comment les composants individuels interagissent entre eux grâce à Redis. La suite de cette section explorera en détail chaque composant et son rôle au sein du système.

3.2.2.2 Client Redis

L'interaction avec Redis nécessite l'utilisation d'un client spécialisé, conçu pour communiquer efficacement avec le serveur Redis et exploiter pleinement ses fonctionnalités. Cette couche de client permet de traduire les commandes et les requêtes de notre système en opérations que Redis peut comprendre et traiter. Dans notre cas, on utilisera un client en C++, qui est Redis++ .

Redis Plus Plus [8] est une bibliothèque client moderne pour C++ qui permet d'interagir avec le serveur Redis de manière efficace. Elle se distingue des autres bibliothèques clients par sa simplicité d'utilisation tout en offrant une riche palette de fonctionnalités.

L'avantage majeur de Redis Plus Plus est qu'elle offre une interface orientée objet, ce qui facilite grandement l'intégration avec les applications C++ modernes. De plus, elle supporte la plupart des fonctionnalités offertes par Redis, de la gestion des types de données de base comme les chaînes, les listes et les hachages, jusqu'aux fonctionnalités plus avancées comme les streams et le pub-sub.

L'utilisation de Redis Plus Plus dans notre système nous permet de tirer pleinement parti de la puissance de Redis tout en conservant l'efficacité et la lisibilité du code C++. Grâce à cette bibliothèque, nous avons pu simplifier notre codebase, améliorer la performance globale de notre système et rendre notre infrastructure plus robuste face aux défis de montée en charge.

3.2.2.3 Diagramme UML du Client

Le client joue un rôle central dans l'architecture de notre système. Il est l'interface principale entre l'application et la base de données Redis, orchestrant la communication et le transfert de données. La figure 3.2.3 représente le diagramme UML du client :

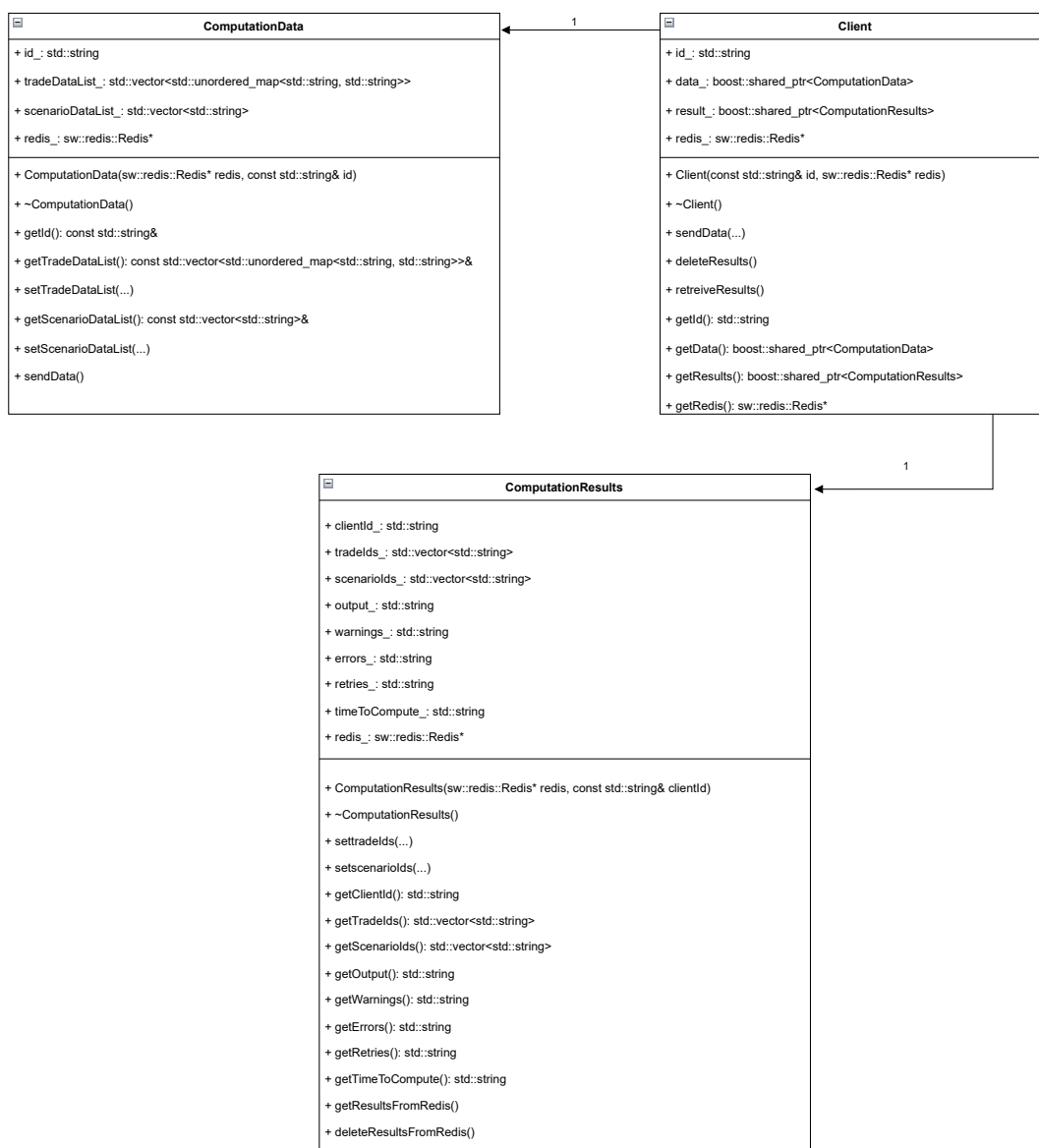


FIGURE 3.2.3 – Diagramme UML du Client

Initialisation et Connexion À la création de l'objet `Client`, une connexion est établie avec le serveur Redis via l'objet `sw::redis::Redis`. Cette connexion est maintenue pendant toute la durée de vie de l'objet, assurant une communication fluide et sans interruption.

ComputationData

- Cette classe stocke les données à traiter. Elle contient deux principales listes : `tradeDataList_` et `scenarioDataList_`, qui stockent respectivement les informations relatives aux trades et aux scénarios. Ces deux listes sont la serealization des objets trades et scenarios.
- La méthode `sendData` permet de pousser ces données dans la base de données Redis. La fonction `sendData()` de la classe `ComputationData` est responsable de la manière dont les données de calcul sont organisées et stockées dans Redis.
 - **Données des trades :** Pour chaque ensemble de données de transaction présent dans `tradeDataList_`, une nouvelle entrée est ajoutée à un Stream identifié par `tradeDataStream:` + l'id du client. Chaque ensemble de données est stocké sous forme de paires clé-valeur au sein de cette entrée.
 - **Données des scénarios :** Les identifiants de scénarios sont ajoutés séquentiellement à une liste Redis sous la clé `scenarioDataList:ClientId`. L'ordre d'insertion des identifiants est maintenu dans cette liste.
 - **ComputationData :** Des informations essentielles concernant les données de calcul sont stockées dans un hash. Cela inclut l'identifiant du client et les clés qui pointent vers les données de transaction et les identifiants de scénarios. Ce hash est accessible via la clé `ComputationData:ClientId`.

L'image suivante 3.3.1 représente un exemple d'un objet `ComputationData` stockées sur Redis ,id du client étant N7 :

Field	Value
ClientId	N7
TradeStreamKey	tradeDataStream:N7
ScenarioListKey	scenarioDataList:N7

FIGURE 3.2.4 – Exemple de stockage de `ComputationData` sur Redis

ComputationResults

- Suite à une opération ou un calcul effectué sur les données, les résultats sont stockés dans cette classe.
- La méthode `getResultsFromRedis` est responsable de la récupération des résultats du serveur Redis. Elle interroge le serveur en utilisant les clés appropriées pour extraire les données et les stocker dans les membres appropriés.
- La fonction `deleteResultsFromRedis` permet de nettoyer la base de données en supprimant les résultats une fois qu'ils ne sont plus nécessaires.

3.2.2.4 Sérialisation des Trades

Le processus de sérialisation des trades est orchestré par la fonction `sendData` de la classe `Client` en suivant les étapes suivantes :

A. Préparation des Conteneurs de Données :

- Une liste `tradeDataList` est définie pour contenir les informations sérialisées de chaque trade. Cette liste est une collection de dictionnaires (ou `unordered_map`), où chaque dictionnaire représente un trade individuel. C'est l'attribut de `ComputationData`.

B. Itération sur les Trades :

- La fonction parcourt ensuite chaque trade.

C. Extraction des Attributs du Trade :

- Pour chaque trade, trois attributs clés sont extraits :
 - `tradeId` : L'identifiant unique du trade.
 - `tradeType` : Le type du trade (par exemple, Swap, Bond, etc.).
 - `tradeXML` : La représentation XML du trade en `std::String`.

D. Création d'un Dictionnaire pour le Trade :

- Ces trois attributs sont ensuite stockés dans un dictionnaire `tradeData` :
 - La clé `"tradeId"` est associée à l'identifiant du trade.
 - La clé `"tradeType"` est associée au type du trade.
 - La clé `"tradeXML"` est associée à la représentation XML du trade.

E. Ajout du Dictionnaire à la Liste :

- Le dictionnaire `tradeData` représentant un trade est ensuite ajouté à la liste `tradeDataList`.

Ainsi, à la fin de cette boucle, la liste `tradeDataList` contient les données sérialisées de tous les trades dans le `book`. Chaque trade est représenté sous la forme d'un dictionnaire, prêt à être stocké ou transmis à Redis pour un stockage ultérieur.

3.2.2.5 Lua Script

Lua est un langage de programmation léger, rapide et intégrable, souvent utilisé pour étendre ou personnaliser le comportement des applications. En ce qui concerne Redis, le support pour les scripts Lua a été introduit pour permettre des opérations atomiques complexes sur les données. Ceci est particulièrement utile pour réaliser des tâches qui, autrement, nécessiteraient plusieurs allers-retours entre le client et le serveur.

Dans le contexte de notre architecture, l'utilisation de deux scripts Lua suggère que des opérations complexes sont effectuées directement sur le serveur Redis, ce qui peut considérablement améliorer les performances en réduisant la nécessité de transférer de grandes quantités de données entre le serveur et le client ou en éliminant la nécessité de plusieurs voyages réseau.

Premier Script : `Task.lua`

Le premier script Lua fournit un exemple de la façon dont nous pouvons manipuler et organiser les données directement sur le serveur Redis, optimisant ainsi la performance et réduisant la charge sur l'application principale.

Ce script commence par récupérer toutes les clés dans `ComputationData`. Il détermine ensuite les clés associées à la liste de scénarios et au stream de trades. Le script va chercher tous les IDs des trades et calcule combien de trades devraient être dans chaque stream segmenté.

La division principale se fait en quatre streams. Pour chaque segment, le script crée un nouveau stream et y ajoute les trades correspondants. Cela est particulièrement utile lorsque nous voulons répartir la charge de traitement sur plusieurs processus ou serveurs.

Après avoir divisé les trades, le script génère ensuite des tâches pour chaque engine ID. Chaque tâche contient des informations telles que l'ID du client, l'état de la tâche, l'ID du processus de l'engine, le temps nécessaire pour effectuer le calcul, et les clés pour le stream des trades et la liste des scénarios. Enfin, un message "Run" est publié pour chaque tâche, indiquant qu'elle est prête à être exécutée.

Deuxième Script : `checkStatus.lua`

Après avoir exécuté des tâches, il est essentiel de surveiller leur état d'achèvement. C'est là qu'intervient le script `checkStatus`. Ce script est conçu pour vérifier périodiquement si toutes les tâches lancées ont été terminées. Plutôt que de consommer des ressources en vérifiant continuellement l'état des tâches, une approche plus efficace est adoptée : une vérification périodique est effectuée à intervalles définis (par exemple, toutes les 5 secondes) pour déterminer si toutes les tâches sont terminées.

Si le script identifie que toutes les tâches sont complétées, un message est affiché pour notifier que l'ensemble des tâches est terminé. Si ce n'est pas le cas, il attend un moment avant de vérifier à nouveau.

Cette approche permet d'assurer que les ressources système ne sont pas gaspillées en vérifications constantes. De plus, elle garantit également que le programme principal est informé dès que toutes les tâches sont complétées, lui permettant ainsi de passer à l'étape suivante ou de terminer son exécution.

3.2.2.6 ORE et son interaction avec Redis

ORE interagit avec Redis en utilisant les Pub/Sub. En effet, le script `Task.lua` suit une stratégie naïve de la distribution des tâches. Si on dispose de N engine ORE, on aura N tâches créées. Ceci pourra être amélioré plus tard.

Pour chaque tâche, on crée un canal portant son nom entre l'engin et Redis. On envoie un message "RUN" pour que ORE récupère les données de sa tâche correspondante.

Des classes ont été ajoutées à ORE pour le permettre récupérer les données et les désérialisées et Redis plus plus a été aussi intégré avec ORE afin de communiquer avec le serveur Redis.

Désérialisation des Trades :

La désérialisation des trades est une étape cruciale pour l'interaction entre ORE et Redis. Ce processus consiste à convertir les trades stockés dans Redis, sous une forme sérialisée, en une forme exploitable pour les traitements ultérieurs par ORE.

Préparation : Avant d'entamer la désérialisation, nous nous sommes assurés que toutes les données relatives aux trades étaient correctement stockées dans Redis. Ces données étaient principalement stockées sous forme de chaînes XML, optimisées pour la transmission et le stockage.

Stratégie adoptée : Une classe spécialisée, nommée `TradeDeserializer`, a été mise en place. Cette classe avait pour objectif principal de servir d'interface entre la base de données Redis et ORE.

- **Connexion à Redis :** À sa création, cette classe établit une connexion à Redis, utilisant les paramètres fournis. Cette connexion est maintenue tout au long de son cycle de vie, permettant un accès rapide et efficace à la base de données chaque fois que cela est nécessaire.
- **Récupération des Données :** En se basant sur l'ID de la tâche fournie, la classe interroge Redis pour obtenir toutes les données sérialisées associées à cette tâche. Les données de chaque trade étaient associées à des clés spécifiques, facilitant leur localisation et leur extraction.
- **Processus de Désérialisation :** Une fois les données sérialisées récupérées, le processus de désérialisation commence. Nous avons utilisé la capacité d'ORE à convertir les représentations XML des trades en objets exploitables. Grâce à une "factory" intégrée à ORE, chaque trade est converti en fonction de son type.
- **Gestion des Erreurs :** Tout au long du processus, nous avons veillé à gérer correctement les erreurs. Si une désérialisation échoue, cela est correctement consigné, et le processus continue, assurant ainsi que quelques erreurs ne bloquent pas l'ensemble du processus.

Résultat Final : À la fin du processus, un ensemble d'objets trade, prêts à être utilisés par ORE.

Désérialisation des Scénarios :

La désérialisation des scénarios, telle qu'elle est mise en œuvre, consiste à convertir des informations stockées dans la base de données Redis en structures de données exploitables. À partir d'un identifiant de tâche spécifié, les informations associées sont récupérées. Si ces données contiennent une clé "scenarioListKey", cela indique l'existence de scénarios liés à la tâche. Ces scénarios sont ensuite extraits séquentiellement et organisés dans un vecteur, prêts pour une analyse ou d'autres traitements ultérieurs dans l'application.

3.2.3 Mise en œuvre de Redis dans notre système - Deuxième tentative

Après un échange avec mon tuteur d'entreprise, il s'est avéré que la première architecture présentait plusieurs défauts :

- **Scalabilité :** La conception initiale ne permettait pas une mise à l'échelle fluide. D'une part, l'ajout d'un nouveau engin dans l'architecture précédente n'aurait aucun effet puisque il n'y aurait aucune tâche à lui attribuer. D'autre part, l'ajout d'un nouveau client pourrait engorger le système, le rendant lent et moins réactif, en raison du manque d'une répartition efficace des requêtes entre les différents engins.
- **Maintenabilité :** Il faut réduire la dépendance à des scripts externes et s'appuyer sur des mécanismes intégrés simplifie la maintenance du système.
- **Sécurité :** Le client ne devrait pas avoir connaissance des détails internes de Redis. Dans la configuration initiale, c'était le client qui déposait directement les données, lui donnant ainsi une compréhension de la manière dont elles étaient structurées sur Redis. Cette exposition pourrait représenter une faille de sécurité, car toute connaissance approfondie de la structure interne facilite les tentatives de manipulations non autorisées.

Face à ces défis, nous avons décidé d'ajouter d'autres fonctionnalités à notre architecture et de remplacer quelques unes.

3.2.3.1 Vue d'ensemble de la 2^{ème} Architecture

La figure .0.2 illustre la refonte de notre architecture, intégrant les composants récemment ajoutés. Par la suite, nous détaillerons les modifications apportées et clarifierons la fonction de chaque nouvelle entité.

3.2.3.2 Store

Le client envoie les données à Redis. Pour chaque envoi de données, Redis exécute un script Lua afin de gérer et générer un identifiant unique pour ces données. Ce script commence par incrémenter une valeur associée à la clé "global :client :id" dans la base de données. Si cette clé n'existe pas encore, elle est créée avec une valeur initiale de 1. Sinon, sa valeur actuelle est augmentée de 1, assurant ainsi que chaque ID est unique. Après avoir généré cet ID, il est ajouté à une liste, "ClientIdsList", pour assurer un suivi. L'ID généré est ensuite renvoyé pour être utilisé dans les étapes ultérieures du traitement des données du client.

3.2.3.3 Task Trigger

Avec l'introduction de fonctions déclenchées (Triggers) dans RedisGears [9], il est possible d'enregistrer une logique répondant aux notifications d'espace de clés Redis, semblables à des procédures stockées activées lors d'actions spécifiques telles que des modifications de données. C'est un système basé sur la notification par évènement.

Un trigger a été créée et se déclenche dès qu'une ComputationData a été créée au sein de Redis. Le trigger dès qu'il l'est déclenchée, récupère computationData et crée alors les tâches.

3.2.3.4 Distribute

Le script Lua *Distribute* a pour objectif de gérer la distribution des tâches. Lors de la création d'une tâche, chaque identifiant de tâche est placé dans une file d'attente de Redis (Queue). Un unique canal Pub/Sub est ouvert et demeure actif jusqu'à la fin des tâches. Chaque moteur se connecte à ce canal et indique sa disponibilité en envoyant un message "Ready" via le Pub/Sub. Ainsi, chaque moteur qui est dans l'état "Ready" reçoit une tâche. Si une tâche n'est pas complétée, elle est remise dans la file d'attente pour être réexécutée.

Listing 3.2 – ore.xml file for Swap NPV label

```
1 local tasks_key_pattern = "tasks*"
2 local pending_tasks = {}
3 local quarantined_tasks = 0
4 -- Fetch all task keys
5 local task_keys = redis.call('KEYS', tasks_key_pattern)
6 for _, key in ipairs(task_keys) do
7     local status = redis.call('HGET', key, 'Status')
8     if status == 'Quarantined' then
9         quarantined_tasks = quarantined_tasks + 1
10    elseif status ~= 'Completed' then
11        local retries = redis.call('HINCRBY', key, 'retries', 1)
12
13        if retries > 3 then
14            redis.call('HSET', key, 'Status', 'Quarantined')
15            quarantined_tasks = quarantined_tasks + 1
16        else
17            redis.call('RPUSH', 'task_queue', key)
```

```

18         end
19     end
20 end
21
22 if #pending_tasks > 0 then
23     redis.call('PUBLISH', 'tasks', 'New task available')
24 end
25 return {#pending_tasks, quarantined_tasks}

```

3.2.3.5 Get&Delete

Ce script est lancé quand le client souhaite récupérer les résultats. Le script envoie au client les résultats stockées dans Results et supprime tous les données liées au client sur Redis.

3.2.4 Diagramme de Séquence

La figure 3.2.5 illustre la séquence d'événements lors de la gestion et de la distribution des tâches dans notre système, en intégrant Redis comme principal intermédiaire. Ce diagramme met en lumière l'interaction entre les différents acteurs, notamment le client, les générateurs de données (Trade et Scenario), ainsi que les scripts Lua (Store, Distribute et Get&Delete) qui opèrent dans Redis et les engins de calculs. L'ensemble du processus vise à optimiser le traitement des tâches, de leur création à leur achèvement.

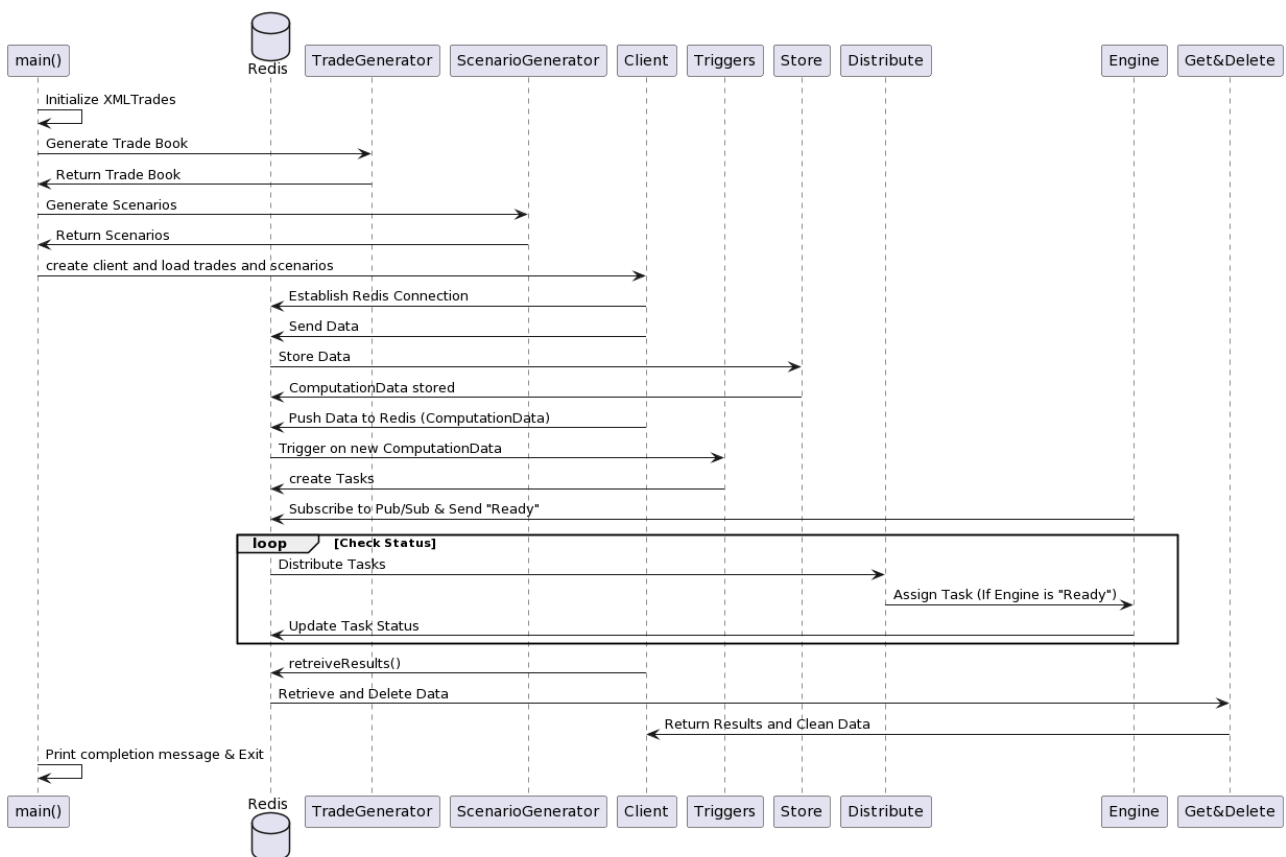


FIGURE 3.2.5 – Diagramme de Séquence end-to-end

3.2.5 Déploiement de l'architecture

3.2.5.1 Engins

L'application `ore_app` est l'un de ces engins. Dans la configuration Docker Compose fournie, cette application est construite à partir d'une image Docker personnalisée, définie par le fichier `Dockerfile-ORE-App`. L'argument `ore_version` spécifie la version particulière de l'application ORE à déployer. Cette application est également connectée à un réseau personnalisé appelé `custom_network`, ce qui lui permet de communiquer avec d'autres services déployés sur le même réseau.

3.2.5.2 Redis

Dans la configuration fournie, nous utilisons une image Docker spécifique, `redislabs/redisgears:edge`, pour déployer une instance de Redis avec le module RedisGears. De même, Redis est connecté au réseau `custom_network`, facilitant la communication avec d'autres services tels que `ore_app`.

Listing 3.3 – Configuration Docker et Dockerfile

```
1 version: '3'
2
3 networks:
4   custom_network:
5     driver: bridge
6
7 services:
8   ore_app:
9     image: ore_app:${ORE_TAG}
10    build:
11      context: ../
12      dockerfile: Docker/Dockerfile-ORE-App
13      args:
14        - ore_version=${ORE_TAG}
15    networks:
16      - custom_network
17  redis:
18    image: redislabs/redisgears:edge
19    ports:
20      - "6379:6379"
21    networks:
22      - custom_network
23
24 ARG ore_version=latest
25 ARG test_tag=latest
26
27 FROM env_ore:${ore_version} as env_ore
28 FROM env_ore_test:${test_tag}
29
30 COPY Examples /ore/Examples
31 COPY Tools /ore/Tools
32 COPY xsd /ore/xsd
33 COPY OREData/test /ore/OREData/test
34 COPY OREAnalytics/test /ore/OREAnalytics/test
35
36 # Update and install required packages
37 RUN apt-get update && apt-get install -y \
38   git \
39   cmake \
40   build-essential \
41   libhiredis-dev
42
```

```
43 # Clone and install redis-plus-plus
44 RUN git clone https://github.com/sewnew/redis-plus-plus.git \
45     && cd redis-plus-plus \
46     && mkdir build \
47     && cd build \
48     && cmake .. \
49     && make \
50     && make install \
51     && cd ..
52
53 # Clean up
54 RUN apt-get clean && \
55     rm -rf /var/lib/apt/lists/* && \
56     rm -rf redis-plus-plus
57
58 RUN mkdir /ore/App
59
60 COPY --from=env_ore /usr/local/bin /ore/App
61
62 RUN true
63
64 COPY --from=env_ore /usr/local/lib /usr/local/lib
65
66 RUN cd /ore \
67     && find -regex ".*\.(sh|txt|in|ac|am)" -exec dos2unix {} ';'
68
69 WORKDIR /ore
70 ENV LD_LIBRARY_PATH=/usr/local/lib:./
71
72 CMD bash
```

3.3 Résultats

3.3.1 Tests

Ces tests ont été effectués sur une machine équipée des caractéristiques suivantes :

- **Processeur** : Intel(R) Core(TM) i9-10885H CPU @ 2.40GHz 2.40 GHz
- **RAM Installée** : 32,0 GB (31,8 GB utilisable)
- **Type de Système** : Système d'exploitation 64 bits, processeur basé sur x64

Remarques :

- Tous les engins montrent une augmentation du temps d'exécution à mesure que le nombre de trades et de scénarios augmente. La tendance générale suggère que des combinaisons de transactions et de scénarios plus complexes nécessitent un temps de calcul plus élevé.
- Un test de 1000 trades et 500 scénarios a été fait sur les 4 engins et a duré 5 min, ce qui est très significatif vu que le même test a duré 47 min pour un seul engin.
- La combinaison de trades et de scénarios "10000 x 5000" représente un goulot d'étranglement de performance significatif pour tous les moteurs, avec une augmentation notable du temps d'exécution.

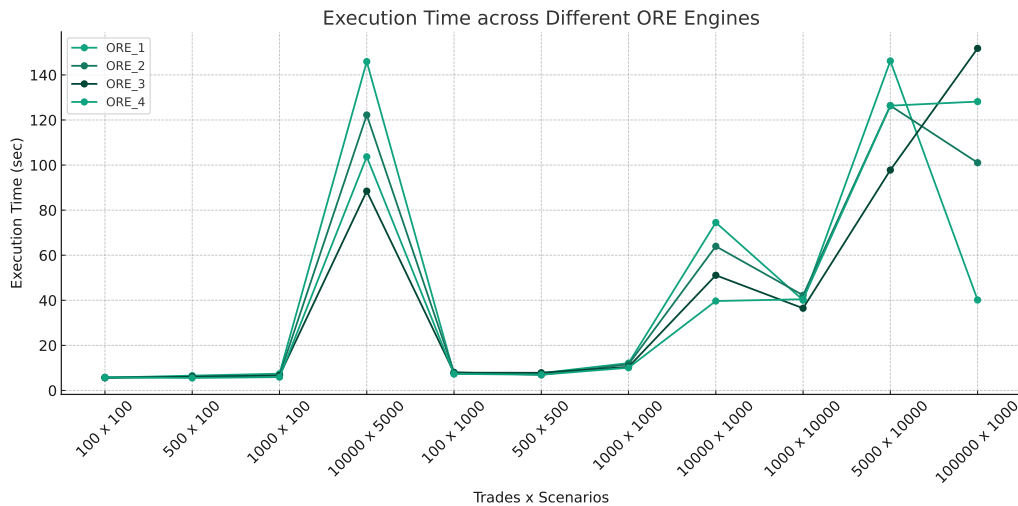


FIGURE 3.3.1 – Testes sur 4 engins et 1 serveur Redis suivant la configuration "Jad"

D'autres testes plus poussées seront réalisés en septembre dès la disponibilité d'une machine du calcul haute performance.

3.3.2 Synthèse et considérations futures

3.3.2.1 Utilisation d'un cluster Redis

Redis, réputé pour ses capacités de stockage de structures de données en mémoire, est devenu la colonne vertébrale de nombreuses applications à forte charge, en particulier dans les scénarios nécessitant un accès rapide à des ensembles de données tels que le caching, le stockage de sessions et l'analyse en temps réel. Cependant, à mesure que les applications se scalent et que la demande augmente, une seule instance Redis peut devenir un goulot d'étranglement, ce qui entraîne une augmentation de la latence et une diminution de la fiabilité. C'est là qu'intervient le cluster Redis.

Un cluster Redis fournit un moyen d'exécuter une installation Redis où les données sont automatiquement partitionnées sur plusieurs nœuds. Ce changement d'architecture introduit plusieurs avantages clés :

- **Échelle horizontale** : Le principal avantage de l'utilisation d'un cluster Redis est la possibilité de scaler horizontalement en ajoutant plus de nœuds. Au lieu de compter sur une seule instance Redis, les données sont réparties sur plusieurs serveurs, ce qui permet au système de gérer plus de connexions réseau et d'opérations par seconde.
- **Redondance des données** : Le cluster Redis offre une haute disponibilité et une redondance des données. Chaque élément de données du cluster est dupliqué sur un nœud secondaire. Si un nœud tombe en panne, son réplica peut servir les données, ce qui garantit qu'il n'y ait pas de point de défaillance unique.
- **Performances améliorées** : La charge est répartie entre plusieurs nœuds Redis, ce qui garantit que les opérations de lecture et d'écriture sont rapides et réduit les chances qu'un seul nœud devienne un goulot d'étranglement en termes de performances.
- **Reprise automatique des pannes** : Si un nœud maître devient indisponible, un nœud esclave correspondant est promu au rôle de maître, ce qui garantit un temps d'indisponibilité minimal.
- **Requêtes distribuées** : Avec un cluster, les requêtes de lecture peuvent être réparties en charge sur les nœuds esclaves, ce qui permet de répartir uniformément la charge de lecture.

3.3.2.2 Utilisation du Machine Learning

Vu qu'on est arrivée à classifier les trades sur ORE par leurs temps d'exécution, on est capable de créer intelligemment les taches.

Un programme basé sur l'apprentissage par renforcement pourra après plusieurs essayées de trouver la stratégie de distribution la plus optimale qui permet d'avoir un temps d'exécution minimal.

Ceci c'est juste une idée qui sera développée plus tard , probablement dans un sujet de stage.

CHAPITRE 4

Conclusion

4.1 Conclusion fonctionnelle

Au cours de mon stage, j'ai exploré les mécanismes complexes de la finance de marché et acquis des connaissances sur des aspects essentiels tels que les transactions(Trades), les données de marché, la calibration et le stress testing.

J'ai maîtrisé l'art de structurer les trades en les reliant de manière complexe aux données du marché. J'ai aussi compris la création de scénario pour le stress test , et comment ça impacte le graphe de dépendance d'un trade.

Essentiellement, cette expérience a enrichi ma compréhension des rouages fondamentaux de la finance de marché, me préparant pour de futurs projets dans ce domaine.

4.2 Conclusion technique

L'objectif du stage du point de vue de l'ENSEEIH est double. Premièrement, il s'agit de comprendre la structure d'accueil, ses mécanismes et ses valeurs. Puis, il doit donner l'occasion de développer des compétences ainsi que d'appliquer dans un environnement professionnel les connaissances acquises à l'école. Je pense avoir atteint ces deux objectifs au travers de mon stage.

Pendant mon stage, j'ai utilisé une large gamme d'outils et de technologies pour exécuter efficacement les tâches et améliorer les résultats du projet. Voici un résumé :

- **Environnement de développement** : J'ai principalement développé en C++ 17 sur l'IDE Visual Studio pour Windows, tout en exécutant sur le Windows Subsystem for Linux (WSL).
- **Gestion de projet** : J'ai utilisé CMake 3.15 pour une gestion de projet efficace et pour rationaliser le processus de construction.
- **Bibliothèques** :
 - *Boost*[11] : J'ai utilisé la bibliothèque Boost version 1.82.0 pour gérer la mémoire de manière dynamique, en garantissant une utilisation optimale des ressources et des performances du système.
 - *Log4cxx*[12] : J'ai utilisé la bibliothèque Log4cxx pour intégrer des mécanismes de journalisation complets, qui ont aidé au débogage et à la surveillance du comportement de l'application.

- *Google Test*[13] : J'ai utilisé la bibliothèque Google Test version 1.11.0 pour faire les tests unitaires.
- **Python** : J'ai utilisé Python 3.8 pour diverses tâches, notamment :
- Lancer des exemples de calcul de NPV avec ORE.
 - Générer des graphes comparatifs pour visualiser les données.
- **Conteneurisation** : J'ai utilisé Docker[10] pour créer des images et faciliter la virtualisation. Cela a été crucial pour garantir un déploiement fluide sur différentes machines et pour maintenir des environnements d'application cohérents.
- **Gestion de Version** : Au niveau de la gestion de version, j'ai utilisé la Git pour mettre de côté temporairement des modifications qui n'étaient pas encore prêtes à être validées en créant des branches sur chaque nouvelles fonctionnalités ajoutées et la validée avant d'ajouter le code.

L'intégration de ces outils et technologies a enrichi mes compétences, me préparant à divers défis en matière de développement et de déploiement de logiciels.

4.3 Conclusion personnelle

Ce stage techniquement très riche est utile pour ma carrière de future. Grâce à la formation que j'ai obtenue à l'ENSEEIH7 et aux connaissances acquises lors de ce stage, je serai sûrement guidé vers le métier de **Software Engineer** spécialisé dans la finance de marché.

J'ai rencontré beaucoup d'obstacles au cours de ce processus. Tout d'abord, apprendre les subtilités de la finance de marché a été un défi, d'autant plus que ces matières n'étaient pas incluses dans le programme de l'ENSEEIH7. De plus, il était difficile de se familiariser avec les bibliothèques financière utilisées dans ce stage. Je n'ai cependant pas hésité à demander de l'aide, surtout à mon manager **Hani SEIFEDDINE**, avec qui j'ai organisé plusieurs réunion tout au long du stage. J'ai pu ainsi manipulés les outils nécessaires.

Je suis fier des capacités que j'ai développées et des défis que j'ai surmontés au cours de cette période. Ce stage m'a donné l'occasion d'en apprendre davantage sur le monde de la finance et de m'immerger dans la culture d'une entreprise renommé comme Murex. J'ai hâte d'utiliser les connaissances que j'ai acquises et les meilleures pratiques dans mes prochains projets.

En conclusion, je souhaite exprimer ma profonde gratitude pour avoir eu l'opportunité d'effectuer mon stage chez Murex. C'est avec un immense enthousiasme que j'anticipe la continuation de mon parcours professionnel au sein de cette entreprise, en tant que **Software Engineer** en CDI.

Table des figures

3.1.1	Étapes de la gestion du risque d'un trade	13
3.1.2	Étapes de la calibration d'un trade	14
3.1.3	Création de scénario d'un trade	15
3.1.4	Esquisse du processus ORE, des entrées et des sorties.	20
3.1.5	Fichier Rapport de l'exemple de swap	24
3.1.6	Pile d'appel de ORE pour calcul d'un NPV sur un Swap	24
3.1.7	Pile d'appel de runAnalytic pour calcul d'un NPV sur un Swap	24
3.1.8	Fichiers orelog.txt sur l'exemple du Swap	25
3.1.9	Temps d'exécution en s pour différents trades sur ORE	26
3.1.10	Architecture de l'engin	28
3.1.11	Résultats de calcul pour 1000 Trades x 500 Scénarios	30
3.2.1	Schéma illustrant le fonctionnement des streams Redis	33
3.2.2	Schéma illustrant le modèle Pub/Sub de Redis	34
3.2.3	Diagramme UML du Client	36
3.2.4	Exemple de stockage de ComputationData sur Redis	37
3.2.5	Diagramme de Séquence end-to-end	42
3.3.1	Testes sur 4 engins et 1 serveur Redis suivant la configuration "Jad"	45
.0.1	1 ^{ère} Architecture en mode distribuée en utilisant Redis	54
.0.2	2 ^{ème} Architecture en mode distribuée en utilisant Redis	55

Liste des tableaux

3.1.1	Comparaison des produits couverts par Strata	17
3.1.2	Liste des trades couverts par Strata	18
3.1.3	Types de trades pris en charge par ORE	21
3.1.4	Valeurs du temps d'exécution pour différents types de trades	23
3.1.5	Classification des types de trades selon le temps d'exécution de BuildPortfolio	27
3.1.6	Pourcentages de la configuration "Jad"	29
3.1.7	Temps d'exécution pour chaque thread	31

Liste des codes

3.1	ore.xml file for Swap NPV label	22
3.2	ore.xml file for Swap NPV label	41
3.3	Configuration Docker et Dockerfile	43

Bibliographie

- [1] MUREX. *MX.3 Platform*. 2023. URL : <https://www.murex.com/en/solutions/mx3-leading-integrated-capital-markets-solution>.
- [2] OPENGAMMA. *Strata Documentation*. URL : <https://strata.opengamma.io/>.
- [3] Open Source RISK. *Open Source Risk*. 2023. URL : <https://www.opensourcerisk.org/>.
- [4] Open Source RISK. *User Guide*. 2023.
URL : <https://www.opensourcerisk.org/wp-content/uploads/2023/04/userguide.pdf>.
- [5] *Open Source Risk Engine - XML Trades*. <https://www.opensourcerisk.org/xml/>.
- [6] Redis LABS. *Introduction to Redis Streams*. 2019.
URL : <https://redis.io/topics/streams-intro>.
- [7] Redis LABS. *Redis Pub/Sub*. 2019. URL : <https://redis.io/topics/pubsub>.
- [8] SEWENEW. *Redis Plus Plus : A Redis client for C++*. 2021.
URL : <https://github.com/sewene/redis-plus-plus>.
- [9] Redis LABS. *Database Trigger Features in Redis*. 2023.
URL : <https://redis.com/blog/database-trigger-features/>.
- [10] Inc. DOCKER. *Docker : Empowering App Development for Developers*.
<https://www.docker.com/>. 2023.
- [11] Boost DEVELOPERS. *Boost C++ Libraries*. <https://www.boost.org/>. 2023.
- [12] The Apache Software FOUNDATION. *Apache log4cxx*.
<https://logging.apache.org/log4cxx/>. 2023.
- [13] GOOGLE. « Google Test ». In : (2023). URL : <https://github.com/google/googletest/>.
- [14] Inc. KITWARE. *CMake : Cross-Platform Make*. <https://cmake.org/>. 2023.

Annexes

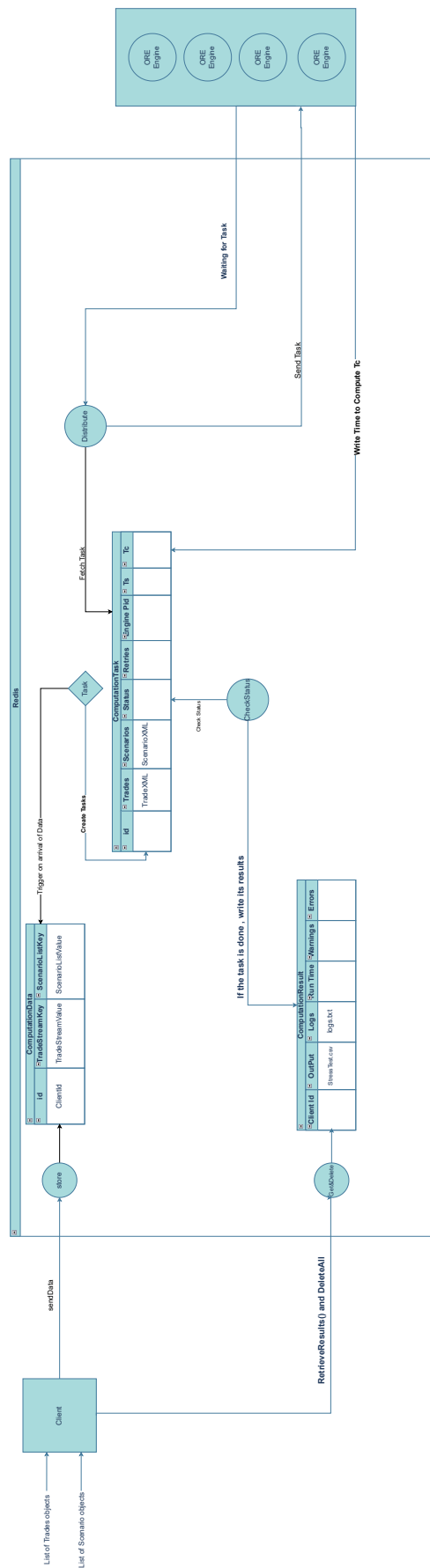


FIGURE .0.2 – 2^{ème} Architecture en mode distribuée en utilisant Redis
55