GRID5000: Make Distribué avec Java RMI

ZARKTOUNI Ismail, GHAZAOUI Badr, MAROUANE Kamal, RIMAOUI Nabila

2023-2024

Introduction

Dans l'ère actuelle de la numérisation et de l'interconnectivité, les systèmes distribués jouent un rôle essentiel dans le fonctionnement quotidien de nombreux services et applications. Leur capacité à coordonner des opérations complexes sur des réseaux étendus et diversifiés est fondamentale pour les performances et la fiabilité de ces services. Notre projet s'inscrit dans cette dynamique innovante, avec objectif de mettre au point et d'évaluer des mécanismes de communication efficaces et robustes pour les systèmes distribués. Au cœur de notre travail, l'utilisation de Java RMI (Remote Method Invocation) se présente comme un choix stratégique, apportant une solution éprouvée pour la communication entre les nœuds répartis sur la plateforme Grid5000. Cette technologie clé permet une intégration transparente et une interaction fiable entre les processus distants, ouvrant des perspectives nouvelles pour la gestion des ressources distribuées et l'optimisation des performances de calcul.

1 Choix Technologiques et Travail Réalisé

1.1 Parsing du Makefile: Analyse lexicale et syntaxique

1.1.1 Analyse Lexicale

L'analyse lexicale dans notre MakefileParser est un processus fondamental qui décompose le texte brut du Makefile en une série de tokens (symboles). Par exemple, les deux-points (':'), les virgules (','), et les tabulations sont identifiés comme des symboles distincts avec des significations particulières dans le contexte d'un Makefile. De plus, cette étape distingue les différentes variables telles que les cibles, les dépendances et les commandes. Les chaînes de caractères sont analysées pour déterminer si elles représentent une cible (précédée d'un deux-points) ou une dépendance (suivie d'une virgule ou d'un espace). Cette analyse lexicale est essentielle pour convertir le texte brut en une série de tokens structurés, facilitant l'interprétation et l'analyse syntaxique qui suit.

1.1.2 Analyse Syntaxique

L'analyse syntaxique de notre MakefileParser joue un rôle crucial dans l'interprétation et la validation de la structure du Makefile. Cette phase s'appuie sur les tokens générés pendant l'analyse lexicale pour construire une représentation logique et structurée du Makefile.

Règles de Production et Fonctions Associées :
Nous avons établi des règles de production spécifiques, transformées en fonctions telles que PROGRAM et RULES dans notre parser. La fonction PROGRAM sert de point d'entrée principal pour l'analyse syntaxique. Elle orchestre le processus en appelant d'autres fonctions comme Rules, qui traitent différentes parties du Makefile en fonction de la structure syntaxique attendue. Cette approche modulaire permet une analyse plus facile et une meilleure maintenance du code.

Représentation des Noeuds et Graphes Orientés :
Chaque noeud dans notre Makefile est représenté en Java par une structure Map<Node, List<Node». Dans cette structure, un Node représente une cible, accompagnée d'une liste de commandes, et la liste de Node associée représente l'ensemble des dépendances de cette cible. Cette architecture nous permet de modéliser le Makefile sous forme de graphe orienté, illustrant clairement les relations entre les cibles et leurs dépendances, comme le montre l'image 1. Cette représentation graphique est non seulement intuitive mais aussi cruciale pour comprendre les interactions complexes entre différents éléments du Makefile.

Vérification des Dépendances Externes :
Une partie essentielle de notre analyse syntaxique est la vérification de l'existence des dépendances externes. Si le parser détecte une dépendance qui n'est pas définie comme une cible dans le Makefile ou qui n'existe pas

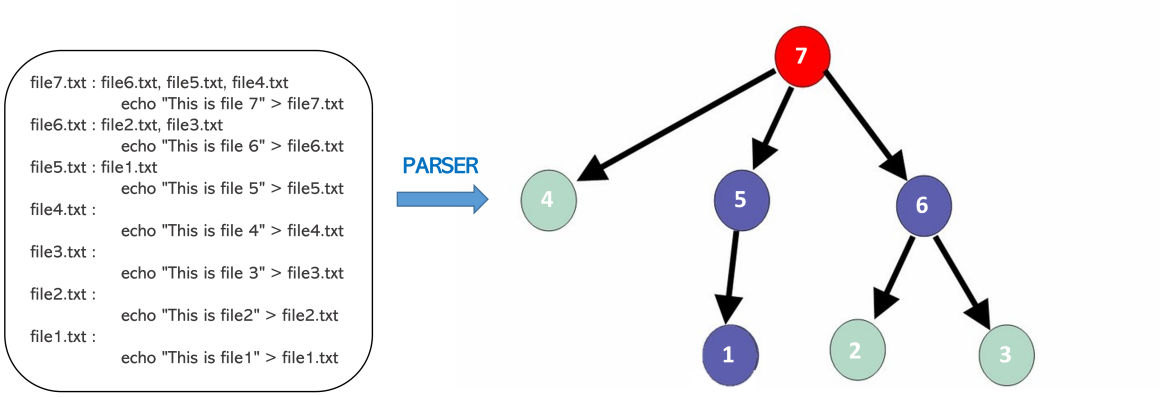

Figure 1: Transformation du makefile en graphe orientée

dans le répertoire courant, il génère une erreur et arrête l'exécution. Cette vérification garantit l'intégrité et la fiabilité du Makefile, s'assurant que toutes les dépendances nécessaires sont accessibles et correctement définies.

1.2 Ordonnancement

L'Ordonnancement est une étape cruciale de notre projet, car il permet de déterminer l'ordre d'exécution de toutes les tâches. La première action consiste à traiter la sortie du parseur du Makefile, qui génère un graphe représentant les cibles et leurs dépendances, ainsi que les commandes à exécuter sur les machines hôtes.

Nous commençons ensuite à parcourir ce graphe en vérifiant si toutes les dépendances d'une cible ont été exécutées, ce qui permettrait alors l'exécution de cette cible. La méthode employée pour cela consiste à récupérer tous les hôtes réservés, puis à appliquer un « **Algorithme de liste** ». Cet algorithme assure que, lorsqu'un hôte est disponible (suivant une logique du premier entré, premier sorti ou FIFO), une tâche lui est attribuée. L'hôte est alors considéré comme occupé jusqu'à ce qu'il ait terminé l'exécution de sa tâche et soit de nouveau disponible. Ce processus est répété pour tous les hôtes. Si une tâche est prête à être exécutée mais aucun hôte réservé n'est libre (tous étant occupés), nous envisageons deux possibilités : attendre (la tâche attendra qu'un hôte se libère) ou réessayer après un délai.
Ensuite, nous devons transférer tous les fichiers nécessaires du maître aux travailleurs qui en auront besoin pour exécuter les commandes. Juste après l'exécution des commandes sur la machine du travailleur, celui-ci renvoie le fichier qu'il a créé en utilisant les commandes.

Concernant la gestion des fichiers, nous envoyons uniquement les fichiers qui n'existent pas déjà sur la machine hôte. Cette approche vise à réduire le temps d'exécution et à permettre la réalisation d'autres tâches plus importantes que l'envoi d'un fichier déjà existant.

1.3 Deploiement sur Grid5000

1.4 Déploiement de la communication RMI de base

Au cours de cette première étape, l'accent a été mis sur l'implémentation de la communication Java RMI entre le nœud maître et les nœuds ouvriers. Cette communication était essentielle pour établir un canal efficace au sein du système distribué.

En se connectant au site via SSH, le script automatisé (./deploy/automate.sh) a été exécuté pour mettre en place la communication RMI. Il est important de souligner que ce script doit être lancé depuis le frontend du site souhaité pour garantir un déploiement correct.

Les résultats de cette étape, enregistrés dans les fichiers master_output.log et node_output.log, étaient facilement accessibles depuis n'importe quelle machine du site grâce au partage NFS commun.

1.5 Déploiement Monosite

Au cours de cette phase, notre objectif était d'établir le système fonctionnel en intégrant la communication réseau déjà mise en place au niveau du parser et de l'ordonnanceur. Pour ce faire, l'exécution d'un makefile spécifique a été réalisée en tant que test de départ.

La première étape consistait à intégrer la communication réseau, préalablement implémentée, au sein du parser et de l'ordonnanceur. Cette action était cruciale pour assurer une coordination efficace entre les différentes composantes du système distribué. La communication RMI précédemment établie entre le nœud master et les nœuds workers a été incorporée dans le flux de données du parser et de l'ordonnanceur.

En tant que validation du déploiement monosite, l'exécution d'un makefile spécifique a été entreprise. Ce test initial visait à vérifier la cohérence du système fonctionnel sur un site unique de Grid5000. Il est important de noter qu'à cette étape, l'envoi de fichiers n'avait pas encore été implémenté, et le déploiement s'effectuait entièrement sur le site local.

Cette phase de déploiement monosite, bien que ne comportant pas encore l'envoi de fichiers, a jeté les bases pour la vérification du bon fonctionnement du système dans un environnement contrôlé. Les résultats de cette étape étaient accessibles sur tout le site Grid5000 grâce au partage NFS commun, facilitant ainsi l'analyse des performances et la détection d'éventuels problèmes.

1.6 Déploiement Inter-Site : API Python

Dans cette troisième phase du projet, le déploiement sur plusieurs sites a été abordé en utilisant une API Python dédiée à Grid5000. Cette approche a été privilégiée pour accroître la flexibilité dans la réservation de nœuds sur différents sites.

La première étape consistait en l'interrogation de l'API de Grid5000 à l'aide de Python. Cette approche a offert une souplesse accrue dans la réservation de nœuds, permettant une gestion plus dynamique des ressources sur plusieurs sites. L'utilisation de Python pour interagir avec l'API a facilité la mise en œuvre de scénarios de réservation plus complexes, adaptés aux besoins spécifiques du déploiement inter-site.

Pour optimiser le processus de réservation, une stratégie de réservation en parallèle a été implémentée. Cela a permis de réserver et d'envoyer les fichiers nécessaires simultanément à plusieurs nœuds sur différents sites. Cette approche parallèle a contribué à réduire le temps nécessaire pour préparer les nœuds sur les sites respectifs, améliorant ainsi l'efficacité globale du déploiement inter-site.

L'introduction de l'envoi de fichiers est devenue nécessaire à cette étape, car les différents sites ne partageaient pas le même système de fichiers. Ainsi, l'implémentation de l'envoi de fichiers a permis de garantir que les fichiers nécessaires au fonctionnement du système étaient correctement transférés vers les nœuds réservés sur chaque site. Cette amélioration a assuré la cohérence des configurations sur l'ensemble des sites, tout en prenant en compte les spécificités de chaque système de fichiers local.

Cette approche de déploiement inter-site, enrichie par l'utilisation de l'API Python de Grid5000, a offert une flexibilité et une optimisation accrues dans la gestion des ressources sur plusieurs sites, renforçant ainsi la robustesse du système distribué déployé.

1.7 Adaptation de l'Exécution au Makefile 'Premier' (Version Monosite)

Cette quatrième phase du projet s'est concentrée sur l'adaptation du système au makefile "premier", une étape essentielle pour évaluer les performances d'exécution du système distribué sur un site unique.

L'objectif principal était d'adapter le système au makefile "premier" afin de mesurer les performances d'exécution. Cette adaptation a impliqué des ajustements spécifiques dans le traitement des tâches définies dans le makefile, prenant en compte les caractéristiques et les exigences particulières du système distribué.

L'objectif était d'optimiser l'exécution des tâches, de maximiser l'utilisation des ressources disponibles, et de garantir une exécution fluide sur le site de déploiement monosite.

2 Mesure de performances brutes et Modélisation du système

2.1 Mesure de la latence et du débit

Dans le cadre de notre analyse approfondie des méthodes de transfert de fichiers, nous avons entrepris une série de mesures comparatives pour évaluer et comparer les performances de trois protocoles de transfert de fichiers couramment utilisés : NFS (Network File System), SCP (Secure Copy Protocol) et RSYNC. Ces expériences ont été conçues pour fournir des insights détaillés sur l'efficacité, la rapidité et la fiabilité de chaque méthode dans divers scénarios de transfert de fichiers.

2.2 Calcul de la latence - Monosite

La Figure 4 démontre les latences mesurées sur différents sites, obtenues en envoyant répétitivement des paquets d'1 octet entre nœuds de chaque site et en calculant leur latence moyenne. Ces résultats fournissent une compréhension claire de la performance de la connectivité au sein de chaque site étudié.

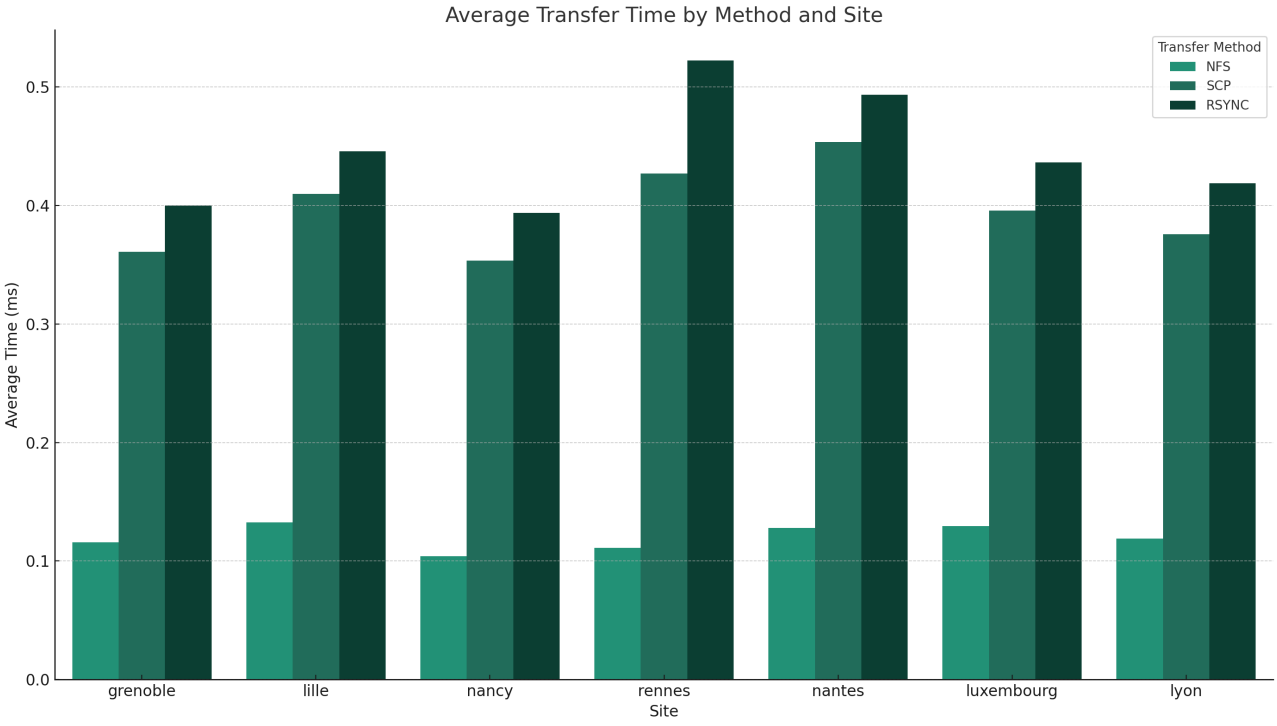

Figure 2: Latence mesurée pour chaque site

Le graphique montre les temps moyens de transfert en millisecondes pour trois différentes méthodes de transfert de données (NFS, SCP et RSYNC) sur plusieurs sites. Voici quelques observations que nous pouvons faire à partir de ce graphique :

Performance de NFS : NFS semble être la méthode la plus rapide pour le transfert de données, avec des temps de transfert considérablement plus courts que les autres méthodes sur tous les sites. Cela indique une efficacité supérieure de NFS dans cet environnement particulier.

Comparaison entre SCP et RSYNC : Les temps de transfert pour SCP et RSYNC sont plus longs que pour NFS, mais ils varient selon les sites. On constate que dans l'ensemble des cas RSYNC a l'avantage par apport à SCP, ce qui suggère que RSYNC pourrait être plus robuste aux variations de réseau. RSYNC est souvent utilisé pour l'efficacité de la synchronisation des données, car il ne transfère que les modifications par rapport aux fichiers existants, ce qui pourrait expliquer une meilleure consistance dans les temps de transfert.

Variabilité entre les sites : Certains sites affichent des temps de transfert plus longs que d'autres pour les mêmes méthodes. Cela pourrait refléter des différences dans l'infrastructure de réseau entre les sites, telles que la bande passante disponible ou la congestion du réseau.

2.2.1 Calcul du débit

L'analyse des données présentées dans le schéma 4 révèle que le site de Nancy se distingue par ses performances exceptionnelles en termes de latence, se positionnant ainsi comme le plus performant parmi tous les sites étudiés. Cette constatation a guidé notre choix de mener une étude approfondie sur le débit spécifiquement sur ce site.

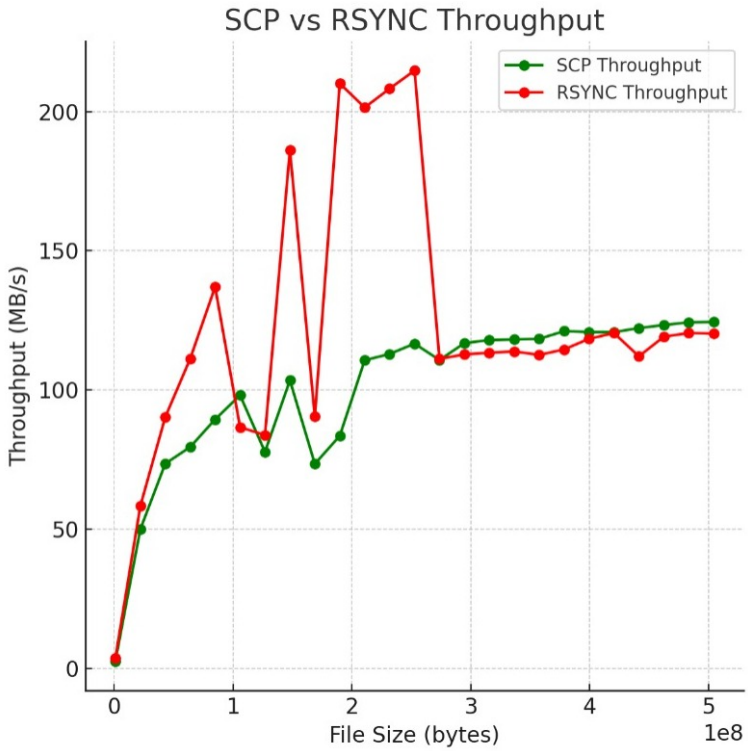

Figure 3: Mesure du débit pour le site de Nancy pour différents tailles de fichiers

Stabilité de SCP : Le protocole SCP, représenté par la ligne verte, montre une performance relativement stable, avec un débit qui augmente légèrement au fur et à mesure que la taille du fichier croît, avant de se stabiliser. Cela indique que SCP maintient une performance constante indépendamment de la taille du fichier.

Variabilité de RSYNC : RSYNC, indiqué par la ligne rouge, présente des pics de performance significatifs, en particulier dans la gamme de tailles de fichiers moyennes à grandes. Ces fluctuations pourraient être dues à l'optimisation des transferts de données que RSYNC tente de réaliser en ne transférant que les parties modifiées des fichiers, ce qui peut conduire à une performance inégale selon le type de données et leur répartition dans les fichiers.

Comportement à grande échelle : Alors que la taille des fichiers augmente, SCP montre une tendance à maintenir son débit, tandis que RSYNC subit des fluctuations plus marquées. Cela pourrait indiquer que RSYNC est plus affecté par les facteurs environnementaux ou par la structure des fichiers lorsqu'ils sont plus volumineux.

Efficacité par rapport à la taille du fichier : Malgré sa variabilité, RSYNC semble parfois atteindre des débits plus élevés que SCP, ce qui pourrait être avantageux pour certains transferts de fichiers.

Implications pratiques : SCP pourrait être préféré pour sa prévisibilité et sa stabilité, en particulier dans des environnements où la performance constante est essentielle. RSYNC pourrait être avantageux pour certaines situations où ses pics de performance dépassent ceux de SCP, mais il faudrait être prêt à gérer sa variabilité.

2.3 Calcul de la latence - Inter-site

Parallèlement à notre analyse focalisée sur un seul site, nous avons élargi notre étude pour inclure l'évaluation des performances de latence entre différents sites. Adoptant une approche méthodique, chaque site a été désigné tour à tour comme 'site maître'. À partir de ce point central, nous avons initié l'envoi de multiples fichiers vers les autres sites. Cette démarche nous a permis d'obtenir une vue globale et comparative des temps de latence inter-sites, offrant ainsi une compréhension plus complète des dynamiques de réseau et des performances de communication à travers divers emplacements géographiques.

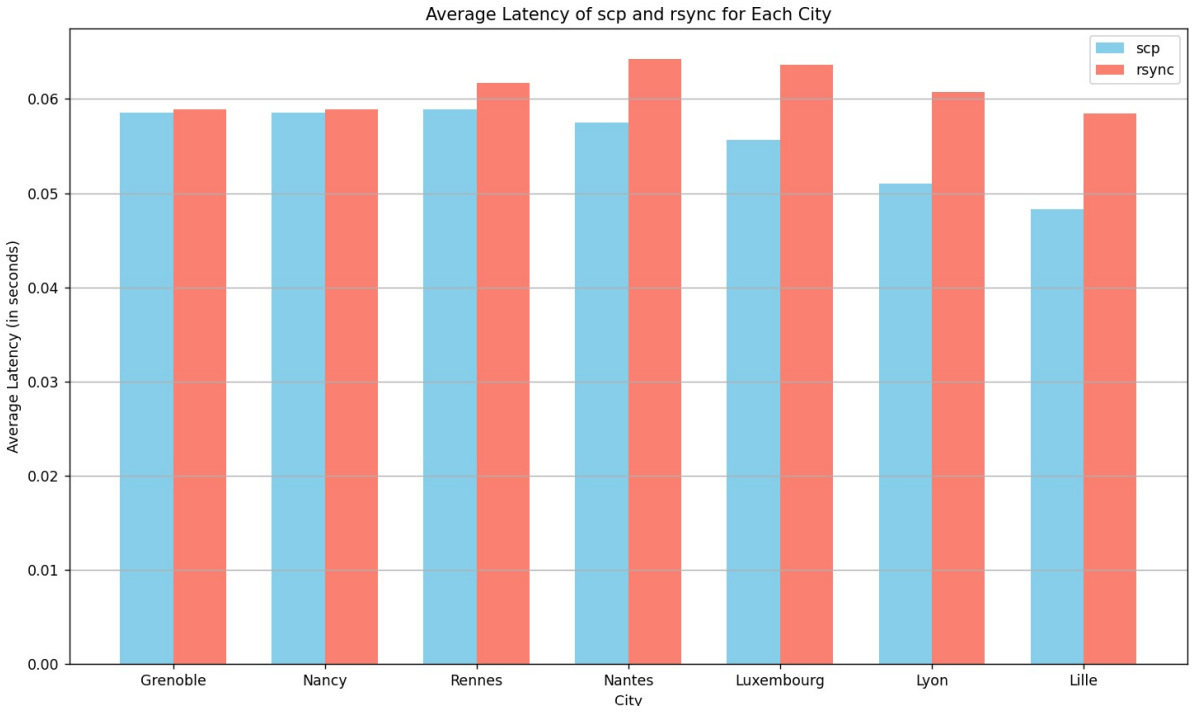

Figure 4: Latence inter-site mesurée pour chaque site considéré comme mast

Variabilité de la latence : Il y a des variations dans la latence pour les différentes villes, ce qui indique que la distance géographique, la topologie du réseau et la configuration du site peuvent affecter la latence des transferts de fichiers.

Comparaison entre SCP et RSYNC : Dans la plupart des sites, RSYNC a une latence plus élevée comparée à SCP. Cela pourrait être dû au surcoût de l'algorithme de RSYNC, qui vérifie les différences de fichiers avant d'effectuer le transfert, tandis que SCP transfère simplement le fichier en entier sans pré-vérification.

Performance de RSYNC : Malgré sa latence plus élevée, RSYNC peut toujours être préférable dans des scénarios où la réduction du volume de données transférées est cruciale, comme lors de la synchronisation de fichiers où seules de petites parties ont changé.

Latence la plus faible : Grenoble et Nancy semblent avoir la latence la plus basse pour les transferts SCP, ce qui pourrait suggérer que ces sites ont une meilleure connectivité ou moins de congestion de réseau.

Considérations pour le choix de la méthode : Le choix de la méthode de transfert peut dépendre de la priorité entre la rapidité (SCP pour les transferts ponctuels et immédiats) et l'efficacité des données (RSYNC pour les mises à jour et synchronisations répétées).

2.4 Mesure d'échange Ping-Pong

2.4.1 Protocole de Mesure

Au sein d'un unique site de la plateforme Grid5000, nous avons mené une série de mesures pour évaluer la latence de communication entre un noeud maître et plusieurs noeuds. Le site unique a été choisi pour éliminer

les variables de latence réseau entre différents sites et se concentrer sur les performances intrinsèques du système.

Dans notre configuration expérimentale, un des noeuds a été désigné comme maître, assumant la coordination des tests et l'initiation des requêtes ping-pong vers les autres noeuds. Ces derniers, exécutant le rôle de nœuds, répondaient aux sollicitations du maître, simulant ainsi un échange de messages au sein d'un système distribué.

À la suite des sessions de test, le noeud maître recueillait les temps de réponse de chaque noeud participant. Ces mesures individuelles étaient ensuite agrégées pour calculer une moyenne représentative de la latence observée dans les échanges ping-pong intra-site.

2.4.2 Résultat des Tests

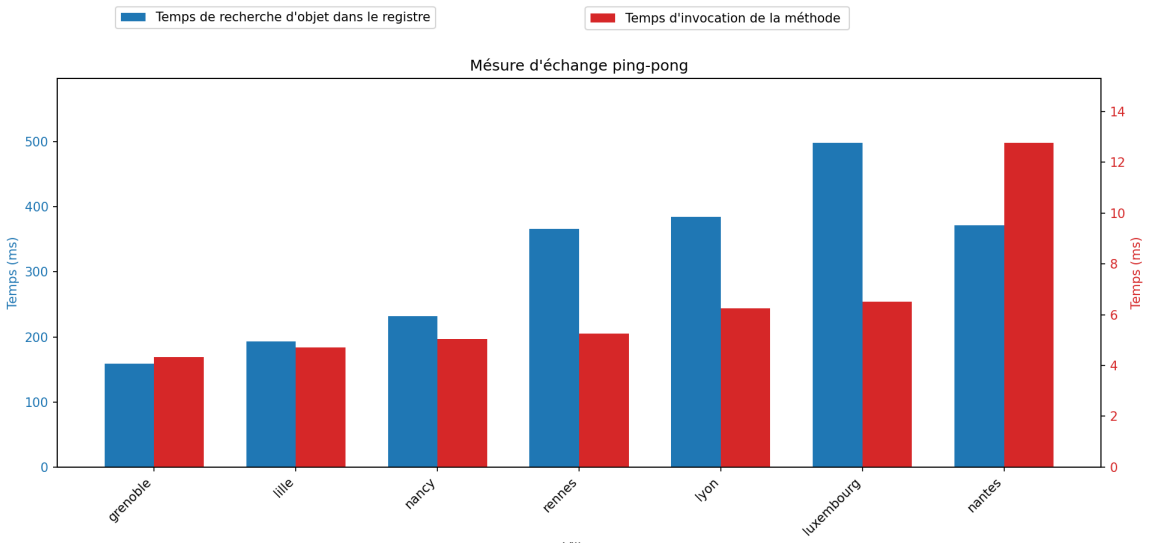

Figure 5: Mesure d'échange Ping-Pong

2.4.3 Étude du Graphe

Le graphique montre clairement deux mesures : le temps de recherche dans le registre et le temps d'invocation de la méthode ping-pong. Nous observons que le temps de recherche est supérieur à celui de l'invocation, ce qui pourrait indiquer que les interactions avec le registre sont plus coûteuses en termes de temps que le simple échange de messages ping-pong.

Il est intéressant de noter que les temps de réponse varient en fonction du site désigné, ce qui suggère que la localisation du site a un impact significatif sur la performance globale. Cela met en évidence l'importance de la sélection stratégique des sites dans le cadre de la conception de systèmes distribués.

2.4.4 Conclusion

Cette analyse révèle l'influence de la topologie du réseau et de la sélection des sites sur les performances de latence dans un système distribué. Les résultats mettent en lumière les défis associés à la gestion des registres distribués et l'importance d'optimiser les opérations de recherche pour améliorer les performances globales.

2.5 Modélisation du système: Makefile "premier"

2.6 Modèle

2.6.1 Graphe d'éxecution

Le contenu du Makefile est le suivant :

```
list.txt: list1.txt list2.txt list3.txt list4.txt list5.txt list6.txt list7.txt list8.txt list9.txt list10.txt list11.txt list12.txt list13.txt list14.txt list15.txt list16.txt list17.txt list18.txt list19.txt list20.txt

    cp list1.txt list.txt ; cat list2.txt » list.txt ; cat list3.txt » list.txt ; cat list4.txt » list.txt ; cat list5.txt » list.txt ; cat list6.txt » list.txt ; cat list7.txt » list.txt ; cat list8.txt » list.txt ; cat list9.txt » list.txt ; cat list10.txt » list.txt ; cat list11.txt » list.txt ; cat list12.txt » list.txt ; cat list13.txt » list.txt ; cat list14.txt » list.txt ; cat list15.txt » list.txt ; cat list16.txt » list.txt ; cat list17.txt » list.txt ; cat list18.txt » list.txt ; cat list19.txt » list.txt ; cat list20.txt » list.txt ;

list1.txt: premier

    ./premier 2 ‘echo 1*200000000/20-1 |bc‘ > list1.txt

list2.txt: premier

    ./premier ‘echo 1*200000000/20 |bc‘ ‘echo 2*200000000/20-1 |bc‘ > list2.txt

...

list20.txt: premier

    ./premier ‘echo 19*200000000/20 |bc‘ ‘echo 20*200000000/20-1 |bc‘ > list20.txt
```

Son graphe d’execution étant :

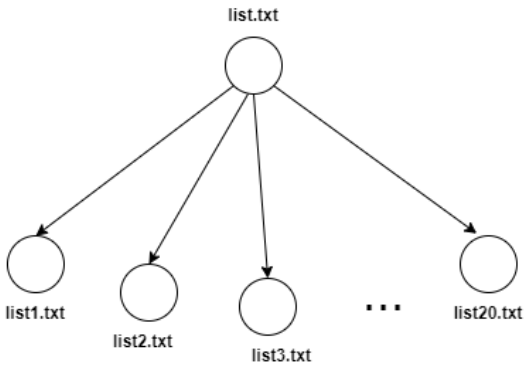

Figure 6: Graphe d’éxecution du Makefile "premier"

2.6.2 Hypothèses

Les données brutes préalablement mesurées serverons de calibres pour ce modèle, et feront l’objet d’hypothèses: Le débit est de 125 MB par seconde, la latence est de 60 micro-secondes

Nous supposerons de plus un nombre d’autres hypothèses :

- Nous nous disposons de n workers
- Le temps pris en compte est celui de l’execution du Makefile, en regroupant tout ce qui n’est pas inclus dans l’execution théorique (parser, temps d’execution du code, ...) dans une constante C.
- L’execution est parfaite : toutes les commandes prennent exactement le meme temps pour s’executer, qui est de 5 secondes. Pour les n workers, à chaque iteration, l’execution des n commandes correspondantes se fait en simultané
- Tous les fichiers crées sont de meme taille 1MB
- le fichier executable "premier" est de taille 1KB

2.6.3 Schéma d'éxecution

Tout d'abord, le fichier executable "premier" est envoyé à tous les workers en parrallèle :

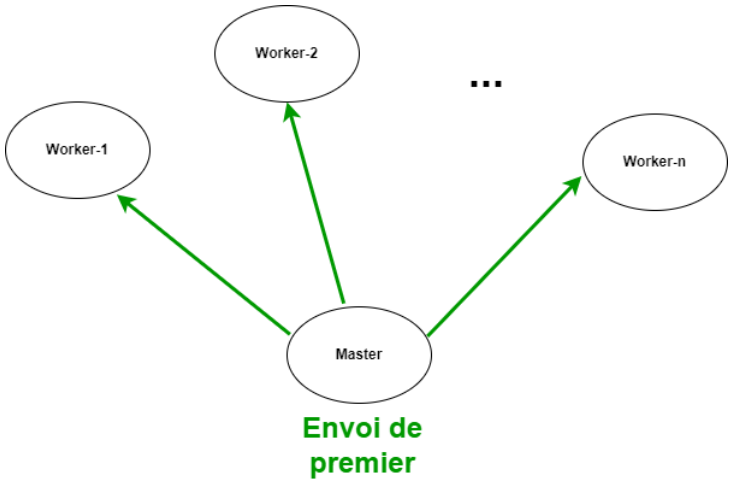

Figure 7: Envoi du fichier éxecutable "premier"

Le temps pris par cette opération est de : $n * (\frac{taille(premier)}{dbit} + 2 * latence)$

Ensuite, les n premières commandes seront éxecutées en parrallèle pendant un temps de 5 secondes.

Figure 8: Execution des commandes

Les n fichiers resultats seront transmis en parrallèle au master : $n * (\frac{taille(listx.txt)}{dbit} + 2 * latence)$

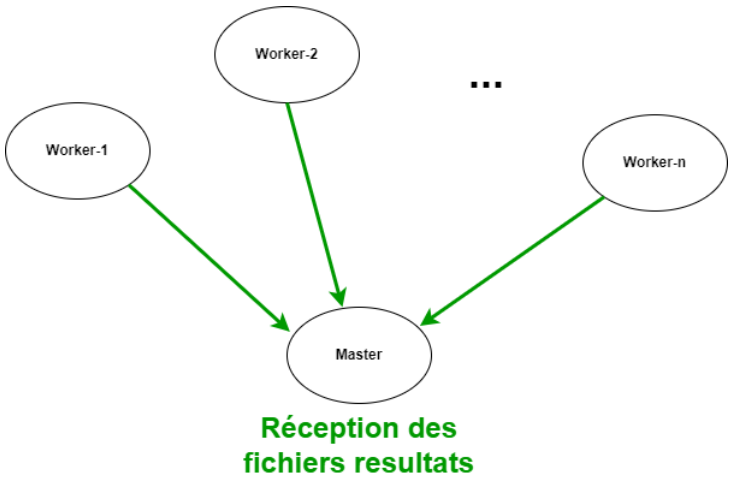

Figure 9: Envoi des fichiers créés

Soit q , r respectivement le quotien et le reste de la la division euclidienne de 20 par n. Ce processus sera répété q+1 fois. Dans la dernière étape, on envoyera que r fichier resultats. L'envoi de "premier", lui, ne se répétera pas. On obtient jusqu'à maintenant la relation suivante :

$$n * (\frac{taille(premier)}{dbit} + 2 * latence) + (q+1) * 5 + (q * n + r) * (\frac{taille(listx.txt)}{dbit} + 2 * latence)$$

soit $n * (\frac{taille(premier)}{dbit} + 2 * latence) + (q+1) * 5 + 20 * (\frac{taille(listx.txt)}{dbit} + 2 * latence)$

A la fin de la dérnière étape, et pour éxecuter la tache "list.txt", le master envoi les fichiers manquants au noeud qui va tout fusionner, soit dans le pire des cas 20-q fichiers. On néglige le temps de la fusion. Ce noeud renvoiyera un fichier final dont la taille est 20 fois celle des fichiers fusionnés.

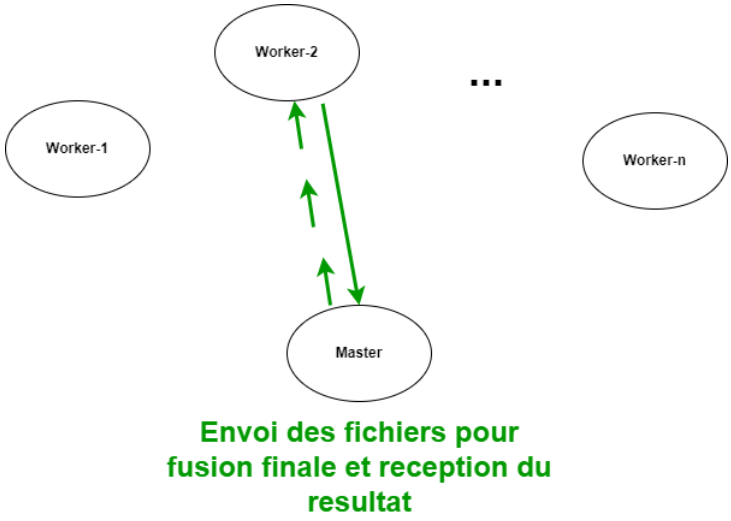

Figure 10: Fusion des fichiers créés

Cette opération dure : $(20 - q) * (\frac{taille(listx.txt)}{dbit} + 2 * latence) + 20 * (\frac{taille(listx.txt)}{dbit} + n * 2 * latence)$

2.7 Simulation

De ce qui précede, on déduit que notre modèle pour le temps d'éxecution global est le suivant :

$$n * (\frac{taille(premier)}{dbit} + 2 * latence) + (q+1) * 5 + (60 - q) * (\frac{taille(listx.txt)}{dbit} + n * 2 * latence)$$

En variant n, nous obtenons le graph suivant :

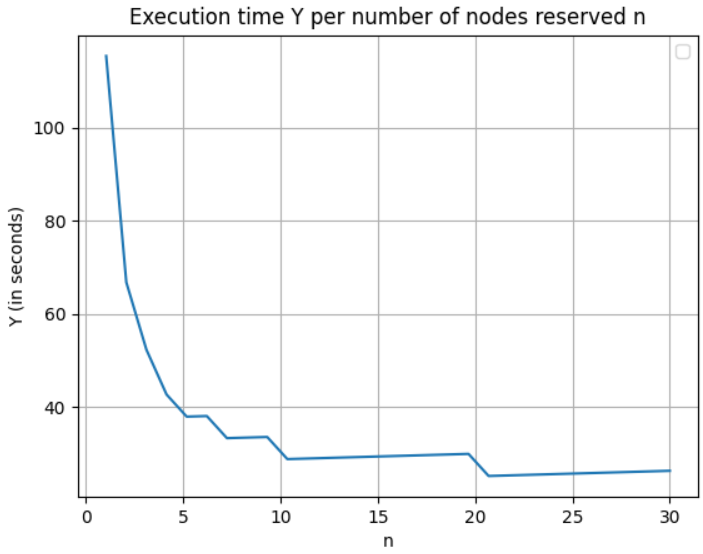

Figure 11: Graphe simulant le modele

3 Analyse et Tests de performance

3.1 Mesure de Performance de l'Exécution : Monosite

Au sein de cette phase, nous avons entrepris une analyse approfondie des performances d'exécution du système sur un site unique, en mettant l'accent sur la variabilité du nombre de nœuds réservés. Les détails de cette mesure de performance sont les suivants.

3.1.1 Démarche de Mesure et Observations

L'exécution du makefile "premier" a été réalisée sur un seul site en faisant varier le nombre de nœuds réservés (`nodes_per_site`). À chaque itération, le temps d'exécution global a été enregistré.

Une observation significative a émergé lorsque le nombre de nœuds réservés atteint le nombre maximal de tâches à exécuter en parallèle, tel que défini dans notre modèle d'exécution. À ce stade, la performance du système tend à stagner, ce qui est cohérent avec les limitations imposées par la capacité d'exécution simultanée de tâches.

3.1.2 Comparaison avec la Simulation du Modèle

La comparaison entre la courbe obtenue après l'expérimentation et celle obtenue par la simulation du modèle a révélé des similitudes frappantes. En particulier, l'aspect de stabilisation après l'atteinte d'un nombre suffisant de nœuds présentait des similitudes. Cependant, des divergences dans les valeurs ont été constatées, démontrant l'existence de facteurs non pris en compte lors de la simulation. La simulation initiale supposait une exécution parfaite, tandis que les résultats réels révèlent des nuances qui peuvent être attribuées à des conditions réelles imprévues.

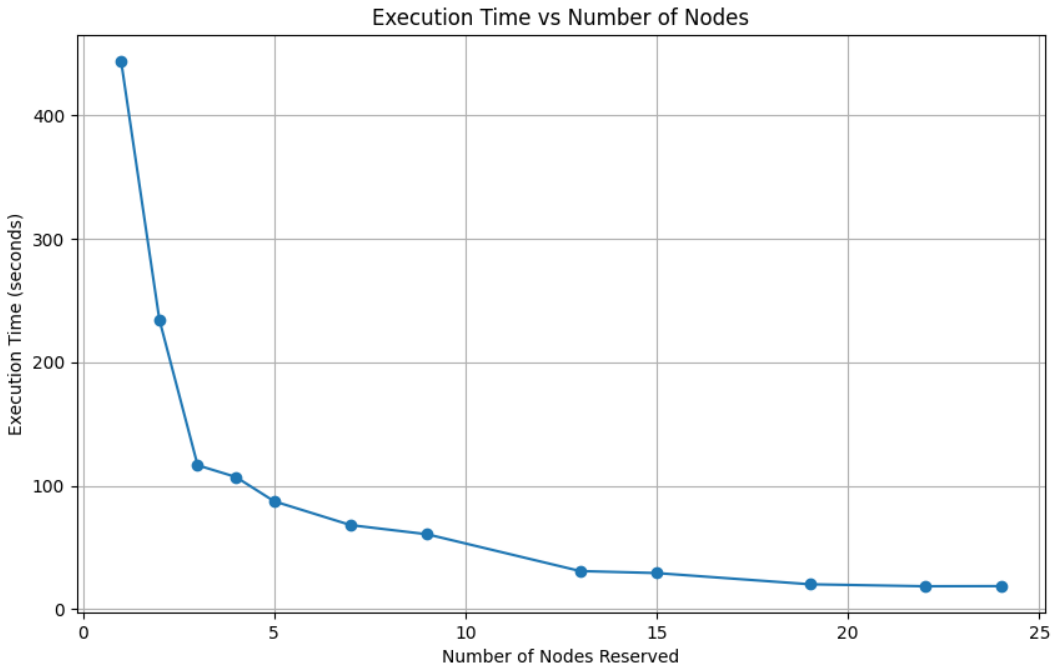

Figure 12: Graphe d'éxecution de l'experimentation

Cette mesure de performance a permis une compréhension plus approfondie du comportement du système lors de son déploiement monosite. Les résultats concrets obtenus ont contribué à valider notre modèle initial tout en mettant en lumière des aspects réels non pris en compte dans la simulation, offrant ainsi des pistes d'amélioration pour des déploiements futurs.

4 Perspectives d'améliorations

4.1 Choix d'une nouvelle architecture autre que master-workers

4.1.1 Architecture Décentralisée Peer-to-Peer (P2P)

Nous proposons une architecture Peer-to-Peer (P2P) comme alternative à l'architecture Master-Workers. Dans le système P2P, chaque nœud fonctionne à la fois comme client et serveur, partageant équitablement les responsabilités de traitement et de communication entre tous les nœuds.

Mise en Œuvre de l'architecture :

- Répartition Équilibrée des Tâches : Dans la conception, chaque nœud gère ses propres tâches et communique directement avec d'autres nœuds pour la distribution et l'exécution des tâches.
- Communication en Réseau Maillé : L'adaptation d'une communication non centralisée, où chaque nœud peut transmettre des informations à plusieurs autres, formant un réseau maillé.
- Gestion Distribuée des Ressources : Les ressources seront gérées de manière décentralisée, chaque nœud prenant des décisions basées sur son état local et les informations partagées par les autres nœuds.
- Algorithme de Découverte et de Routage : l'implémentons des algorithmes pour la découverte de nœuds et le routage efficace des requêtes et des données à travers le réseau.

4.1.2 Comparaison de Performances Entre l'Architecture Master-Workers et l'Architecture P2P

Pour évaluer les performances entre notre architecture Master-Workers actuelle et la nouvelle architecture P2P proposée, nous devons considérer plusieurs critères clés :

4.1.2.1 Efficacité de l'Utilisation des Ressources

- Efficacité de l'Utilisation des Ressources Master-Workers : Les ressources peuvent être sous-utilisées en raison de la centralisation des décisions et de la distribution des tâches. Risque de surcharge du nœud maître, entraînant des goulots d'étranglement.
- P2P : Utilisation optimisée des ressources, chaque nœud prenant en charge des tâches adaptées à ses capacités. Moins de risque de surcharge sur un seul nœud, réduisant les goulots d'étranglement.

4.1.2.2 Scalabilité et Flexibilité

- Master-Workers : La scalabilité peut être limitée par la capacité du nœud maître à gérer un grand nombre de workers. Moins flexible en réponse aux changements rapides de charge de travail.
- P2P : Meilleure scalabilité, car l'ajout de nouveaux nœuds n'accroît pas la charge sur un nœud central. Plus flexible, car les nœuds peuvent rejoindre ou quitter le réseau sans perturber l'ensemble du système.

4.1.2.3 Résilience et Tolérance aux Pannes Master-Workers :

- Moins résilient en cas de défaillance du nœud maître. La panne d'un worker a un impact limité sur l'ensemble du système.
- P2P : Plus résilient, car la défaillance d'un nœud a moins d'impact sur l'ensemble du réseau. La redondance et la distribution des tâches améliorent la tolérance aux pannes.

4.1.2.4 Latence et Performance de Communication Master-Workers :

- La communication centralisée peut introduire de la latence, surtout dans un environnement à grande échelle. La performance peut être affectée par la congestion du réseau autour du nœud maître.
- P2P : Réduction potentielle de la latence grâce à des communications directes entre les nœuds. Moins susceptible de rencontrer des problèmes de congestion réseau.

4.1.2.5 Gestion et Complexité de l'Ordonnancement Master-Workers :

- Ordonnancement centralisé plus simple à gérer mais peut être moins efficace. Manque de flexibilité dans l'attribution dynamique des tâches.
- P2P : Ordonnancement plus complexe mais permet une attribution des tâches plus dynamique et adaptative. Peut nécessiter des algorithmes sophistiqués pour une gestion efficace.

4.2 Choix d'un Nouvel Algorithme d'Ordonnancement

Dans notre projet actuel, l'algorithme d'ordonnancement utilisé est basé sur un « Algorithme de liste », suivant une logique du premier entré, premier sorti (FIFO). Ce système attribue les tâches aux hôtes disponibles, les considérant occupés jusqu'à la fin de l'exécution de la tâche. Si aucune ressource n'est disponible, la tâche est mise en attente ou réessayée après un délai.

Nouvel Algorithme d'Ordonnancement : Ordonnancement Basé sur les Capacités et les Besoins

4.2.1 Principe

- L'ordonnancement est fondé sur l'évaluation des capacités de chaque nœud et les exigences spécifiques des tâches.
- Il prend en compte non seulement la disponibilité des nœuds mais aussi leur performance, mémoire, bande passante et d'autres caractéristiques techniques pertinentes.

4.2.2 Mécanisme de Décision

- Chaque nœud évalue les tâches en attente et décide de prendre en charge celles pour lesquelles il est le mieux équipé.
- Ce mécanisme assure une meilleure adéquation entre les tâches et les nœuds, optimisant ainsi l'utilisation des ressources.

4.2.3 Implémentation

- Nous utiliserons des mécanismes de communication inter-nœuds pour partager les informations sur les capacités et les besoins.
- Des algorithmes d'optimisation pourront être intégrés pour équilibrer la charge de travail et optimiser les performances globales du système.

4.3 Utilisation du Réseau Haute Performance dans l'Architecture P2P

4.3.1 Communication Directe et Décentralisée :

L'architecture P2P permet une utilisation plus intensive du réseau haute performance grâce à des communications directes et décentralisées entre les nœuds. Chaque nœud peut communiquer simultanément avec plusieurs autres, exploitant ainsi mieux la bande passante disponible.

Réduction de la Latence : En éliminant le besoin d'un nœud central pour relayer les informations, nous réduisons la latence dans la transmission des données. Cela est particulièrement bénéfique pour des tâches nécessitant des échanges rapides et fréquents de données.

4.3.2 Optimisation de la Bande Passante :

- La répartition des charges de communication sur plusieurs nœuds évite la congestion du réseau centrée autour d'un serveur maître.
- Cela permet une utilisation plus homogène et efficace de la bande passante sur l'ensemble du réseau.

4.3.3 Transferts de Données à Haute Vitesse :

- Les capacités du réseau haute performance sont utilisées pour effectuer des transferts de données à grande vitesse entre les nœuds.
- Cela est crucial pour les opérations qui nécessitent un grand volume de données, comme les simulations scientifiques ou le traitement de données massives.

4.3.4 Redondance et Résilience :

- En cas de défaillance d'un nœud ou d'un lien, le réseau P2P peut rapidement reconfigurer les chemins de communication, assurant ainsi une continuité de service.
- Cette redondance améliore la résilience globale du système.

4.3.5 Comparaison avec l'Architecture Master-Workers :

- Master-Workers :

- La communication se fait principalement entre le nœud maître et les workers, ce qui peut entraîner des goulots d'étranglement et une utilisation inégale du réseau.
- La latence est plus élevée en raison de l'intermédiation du nœud maître.

- P2P :

- Meilleure utilisation du réseau grâce à des communications directes entre les nœuds.
- Réduction significative de la latence et meilleure gestion de la bande passante.

Conclusion

En conclusion du rapport sur le projet GRID5000 utilisant Make Distribué avec Java RMI, plusieurs éléments clés se dégagent.

Premièrement, le projet a démontré l'efficacité de l'utilisation de Java RMI dans la mise en œuvre de systèmes distribués sur la plateforme Grid5000. Cette technologie a permis une communication fiable et efficace entre les nœuds distribués, essentielle pour la coordination des opérations complexes.

Deuxièmement, les différentes phases du projet, comprenant l'analyse lexicale et syntaxique du Makefile, l'ordonnancement des tâches, et le déploiement sur divers sites de Grid5000, ont été couronnées de succès. Ces étapes ont montré l'importance d'une approche structurée et méthodique dans la gestion des systèmes distribués.

Troisièmement, les tests de performance et les analyses menées ont fourni des insights précieux sur les protocoles de transfert de fichiers et la latence dans un environnement distribué. Ces résultats soulignent l'importance de choisir les bonnes méthodes et stratégies de communication en fonction des besoins spécifiques du système.

Enfin, les perspectives d'amélioration suggérées, notamment le passage à une architecture Peer-to-Peer et l'introduction de nouveaux algorithmes d'ordonnancement, ouvrent la voie à des développements futurs prometteurs. Ces améliorations visent à optimiser davantage l'utilisation des ressources, la scalabilité, la flexibilité, la résilience et la performance de communication dans les systèmes distribués.

Le projet GRID5000 avec Make Distribué et Java RMI représente donc une avancée significative dans la compréhension et la gestion des systèmes distribués, offrant des pistes solides pour de futures recherches et applications dans ce domaine.