Adaptive Consistency Protocols for Replicated Data in Modern Storage Systems with a High Degree of Elasticity

Sathiya Prabhu Kumar

CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS | le cnam

École Doctorale du Conservatoire National des Arts et Métiers

LABORATOIRE CEDRIC - CNAM
LABORATOIRE LISITE - ISEP

THÈSE DE DOCTORAT

présentée par : **Sathiya Prabhu KUMAR**

soutenue le : **15 Mars 2016**

pour obtenir le grade de : **Docteur du Conservatoire National des Arts et Métiers**

Spécialité : **INFORMATIQUE**

Adaptive Consistency Protocols for Replicated Data in Modern Storage Systems with High Degree of Elasticity

THÈSE dirigée par

M. GRESSIER-SOUDAN Eric — *Professor, CEDRIC-CNAM*

Encadrée par

Mme. CHIKY Raja — *Enseignant-Chercheur, LISITE-ISEP*
M. LEFEBVRE Sylvain — *Enseignant-Chercheur, LISITE-ISEP*

RAPPORTEURS

M. DEFUDE Bruno — *Professor, TELECOM SUDPARIS*
M. HAGIMONT Daniel — *Professor, IRIT/ENSEEIHT*

EXAMINATEURS

Mme. CHABRIDON Sophie — *Assistant professor, TELECOM SUDPARIS*
Mme. RONCANCIO Claudia — *Professor, GRENOBLE INP*

To my supervisor Dr. Raja Chiky
who gave me this opportunity.

Acknowledgments

I owe an indebted gratitude to my supervisor Ass Prof. Raja Chiky, who gave me this opportunity believing in me and put so much effort in finding the funding for the thesis. I was very fortunate to have Prof. Eric Gressier Soudan as my thesis director. He is very kind and caring. He always managed to find some time to make him available for any questions or discussions regarding the thesis. During each meeting, he was like rain of knowledge, gave tons and tons of useful references and valuable ideas to move forward. Words are not enough to thank Ass Prof. Sylvain Lefebvre who gave me a wealth of technical support and constantly pushed me forward regarding the quality of the work. He was very energetic and gave his full dedication for the goodness of the thesis work. Without his support and remarks, this thesis would not have been well accomplished.

I am very thankful to Olivier Hermant for his time and effort in helping me with the formalism part of Chapter 3. I also owe a sincere gratitude to all the team members of RDI - LISITE: Yousara Chabchoub, Zakia Kazi-Aoul, Matthieu Manceny, Mohamed Sellami for being very kind to me and helped me with any question or suggestion regarding the thesis. I am very thankful to Gilles Carpentier for taking care of all the teaching activities that I was involved in these three years and making sure the time and schedule of my teaching activities do not affect the thesis work. I also thank all my ISEP colleagues particularly Ahmad, Manuel, Chen, Navneet, Nuraishah, Amadou and Rayane who took part in all ups and downs of my Phd journey.

I am very grateful to Minyoung Kim and Mark Oliver Stehr for offering me an opportunity to work with them as an International Fellow at Stanford Research Institute, California. The fellowship was very fruitful and filled with lot of learning and experience. Minyoung and Mark are very kind, motivating and are responsible for Chapter 7 of the thesis.

I sincerely thank all the midterm and defense jury members: Prof. Philippe Rigaux, Prof. Bruno Defude, Prof. Daniel Hagimont, Prof. Claudia Roncancio and Ass Prof. Sophie Chabridon for accepting to be a part of the jury and providing their valuable time and cooperation in realizing the defense in time.

I thank Duyhai Doan from Datastax for his support and helpful feedback on the Chapter 7 of the thesis. I also appreciate Datastax and other enterprises, communities, association and individuals for enlightening the world with lot of knowledge and information via tons of websites, blogposts, tutorials, meetups, workshops and conferences.

Last but not least I would like to thank my father for giving me lot of strength, lessons and blessings for facing all the goods and bads throughout my life and making me who I am. The love and affection from my mom and my sister have also been invaluable. Special thanks to all my friends, well wishers and the almighty who stood by me and continue hoping for my best.

I also acknowledge Amazon AWS Research grant for supporting my experimentation and partial support from NSF grant 0932397 for supporting my international fellowship. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the view of NSF.

Résumé

Les systèmes de base de données modernes stockent les objets sous forme de paires de clé et de valeur(s), où chaque valeur est identifiée par une clé unique. Cette approche facilite la distribution des données sur le réseau (grâce aux tables de hachage distribuées) et permet un accès rapide et simple à chaque objet. La plupart de ces systèmes choisissent de privilégier un meilleur niveau de disponibilité et de meilleurs temps de réponse contre une gestion de la cohérence des données. Comme ces systèmes sont des systèmes pair à pair, chaque noeud traite les requêtes en écriture ou en lecture à l'aide de stratégies de réplication optimiste. Il est ainsi possible que des incohérences temporaires entre les éventuelles répliques existent. Ces incohérences peuvent se manifester sous la forme de lectures obsolètes, ou de mises à jour conflictuelles. Le niveau de cohérence obtenu dans ces systèmes dépend du nombre de noeuds contactés à chaque requête. Certains systèmes autorisent l'utilisateur à choisir le nombre de répliques à contacter en fonction du niveau de cohérence attendu et adapte son fonctionnement pour chaque requête ou chaque table. Choisir un haut niveau de cohérence ajoute un coût supplémentaire en terme d'échanges de messages et donc de latence, et risque de provoquer l'échec de la requête s'il n'est pas possible de contacter un nombre suffisant de répliques.

Les principales contributions de cette thèse sont au nombre de trois. La première partie de cette thèse concerne le développement d'un nouveau protocole de réplication nommé LibRe, permettant de limiter le nombre de lectures obsolètes dans un système de stockage distribué. LibRe est un acronyme signifiant "Library for Replication". Le principal objectif de LibRe est d'assurer la cohérence des données en contactant un minimum de répliques durant les opérations de lectures et d'écritures. Dans ce protocole, lors d'une opération d'écriture, chaque réplique met à jour un registre (la "librairie"), de manière asynchrone,

avec l'identifiant de version de la donnée modifiée. Lors des opérations de lecture, la requête est transférée au réplica le plus approprié en fonction de l'information figurant dans le registre. Ce mécanisme permet de limiter le nombre de lectures obsolètes. L'évaluation de la cohérence d'un système reste un problème difficile à résoudre, que ce soit par simulation ou par évaluation en conditions réelles. Par conséquent nous avons développé un simulateur appelé Simizer, qui permet d'évaluer et de comparer la performance de différents protocoles de cohérence. Le système d'évaluation de bases de données YCSB a aussi été étendu pour évaluer l'échange entre cohérence et latence dans les systèmes de stockage modernes. Le code du simulateur et les modifications apportées à l'outil YCSB sont disponibles sous licence libre.

Bien que les systèmes de bases de données modernes adaptent les garanties de cohérence à la demande de l'utilisateur, anticiper le niveau de cohérence requis pour chaque opération reste difficile pour un développeur d'application. La deuxième contribution de cette thèse cherche à résoudre ce problème en permettant à la base de données de remplacer le niveau de cohérence défini par défaut par d'autres règles définies à partir d'informations externes. Ces informations peuvent être fournies par l'administrateur ou un service extérieur. Dans cette thèse, nous validons ce modèle à l'aide d'une implémentation au sein du système de bases de données distribué Cassandra. La troisième contribution de cette thèse concerne la résolution des conflits de mise à jour. La résolution de ce type de conflits nécessite de retenir toutes les valeurs possibles d'un objet pour permettre la résolution du conflit grâce à une connaissance spécifique côté client. Ceci implique des coûts supplémentaires en termes de débit et de latence. Dans cette thèse nous discutons le besoin et la conception d'un nouveau type d'objet distribué, le registre à priorité, qui utilise une stratégie de détection et de résolution de conflits spécifique au domaine, et l'implante côté serveur. Notre approche utilise la notion d'ordre de remplacement spécifique. Nous montrons qu'un type de donnée paramètrée par un tel ordre peut fournir une solution efficace pour les applications demandant des solutions spécifiques à la résolution des conflits. Nous décrivons aussi l'implémentation d'une preuve de concept au sein de Cassandra.

Mots clés : NoSql, Théorème CAP, Cohérence éventuelle, Systèmes de Quorums.

Abstract

In modern distributed database systems, data objects are stored as key-value pairs, in which each data object is identified by a unique key. Storing data with a unique key makes it easy to distribute the data over a storage network (e.g., using hashing techniques) and access the data object simply by passing the key. Most of these data stores choose to trade consistency in favor of request latency and availability by letting any replica node handle read and write requests following optimistic replication strategies. These systems let the replicas hold different values for the same key, accepting temporary inconsistency between the replicas. Two major variants of the inconsistency issues are Stale Reads and Update Conflicts. Ensuring strong consistency in these systems depends on number of replica nodes contacted during read and write requests. The system lets the users choose the number of replica nodes to contact depending on the needed consistency level and adapts the consistency level of the system on a per query or per table basis. In addition to an extra cost spent on request latency in contacting a sufficient number of replica nodes, if a sufficient number of nodes is unable to be contacted, the system fails the request, affecting the availability of the system.

The main contributions of this thesis are three folds. The first contribution of the thesis focuses on an efficient way to control stale reads in modern database systems with the help of a new consistency protocol called LibRe. LibRe is an acronym for Library for Replication. The main goal of the LibRe protocol is to ensure data consistency by contacting a minimum number of replica nodes during read and write operations with the help of a library information. According to the protocol, during write operations each replica node updates a registry (library) asynchronously with the recent version identifier of the updated data. Forwarding the read requests to a right replica node referring the

registry information helps to control stale reads during read operations. The evaluation of data consistency remains challenging both via simulation as well as in a real world setup. Hence, we implemented a new simulation toolkit called Simizer that helps to evaluate the performance of different consistency policies in a fast and efficient way. We also extended an existing benchmark tool YCSB that helps to evaluate the consistency-latency tradeoff offered by modern database systems. The codebase of the simulator and the extended YCSB are made open-source for public access. The performance of the LibRe protocol is validated both via simulation as well as in a real setup with the help of extended YCSB.

Although the modern database systems adapt the consistency guarantees of the system on a per query basis, anticipating the consistency level of an application query in advance at application development time remains challenging for the application developers. In order to overcome this limitation, the second contribution of the thesis focuses on enabling the database system to override the application-defined consistency options during run time with the help of an external input. The external input could be given by a data administrator or by an external service. The thesis validates the proposed model with the help of a prototype implementation inside the Cassandra distributed storage system.

The third contribution of the thesis focuses on resolving update conflicts. Resolving update conflicts often involve maintaining all possible values and perform the resolution via domain-specific knowledge at the client side. This involves additional cost in terms of network bandwidth and latency, and considerable complexity. In this thesis, we discuss the motivation and design of a novel data type called priority register that implements a domain-specific conflict detection and resolution scheme directly at the database side, while leaving open the option of additional reconciliation at the application level. Our approach uses the notion of an application-defined replacement ordering and we show that a data type parameterized by such an order can provide an efficient solution for applications that demand domain-specific conflict resolution. We also describe the proof of concept implementation of the priority register inside Cassandra.

Keywords: NoSql Systems, CAP Theorem, Eventual Consistency, Quorum Systems.

Table of Contents

List of Tables

List of Figures

Résumé étendu

Problématique

La quantité croissante d'informations à stocker et traiter a mis en exergue l'importance d'une gestion efficace des ressources de stockage et de calcul. Il est attendu que ce phénomène, avec la mouvance "Big Data" se renforce encore dans le futur. Ce déluge de données entraîne la recherche de nouvelles approches pour organiser, gérer et explorer ces grandes quantités d'informations.

Les systèmes de bases de données distribués sont une réponse à cette problématique. Ces systèmes incluent des mécanismes d'échanges permettant de se conformer aux besoins des applications.

La réplication est utilisée dans les systèmes de stockages distribués afin d'améliorer la performance et la fiabilité du système. La réplication est le processus par lequel une donnée est copiée en plusieurs exemplaires répartis sur des machines physiques différentes. Bien que l'emplacement de ces copies soit différent, logiquement chaque copie représente la même information. En fonction des besoins de l'application, une donnée peut être répliquée entièrement ou partiellement, et le nombre et l'emplacement des copies peut varier. Dans le reste de ce document, nous appellerons indifféremment les copies d'une donnée et la machine stockant une copie "répliqua". Le fait de recopier les données introduit immédiatement le problème de la mise à jour des différentes répliques: lorsqu'une copie est modifiée, les autres copies deviennent obsolètes et ces changements doivent être reflétés sur les autres copies. Ce processus permet de maintenir la cohérence des données du système distribué. En cas de mises à jour fréquentes des données, il est nécessaire de faire appel à un protocole de réplication [LAEA95b], permettant de gérer correctement les opération de

création, lecture, mise à jour et suppression des données. Le comportement de ce protocole aura des conséquences sur plusieurs autres propriétés du système telles que : la cohérence, évidement, mais aussi la latence, la disponibilité, le passage à l'échelle et la tolérance aux pannes.

Le théorème CAP (Consistency, Availability, Partition Tolerance) [FGC+97; GL02a] énonce qu'un système distribué ne peut garantir en même temps les trois propriétés que sont la tolérance aux partitions (isolation d'une sous partie des noeuds du système), la cohérence et la disponibilité du système. En revanche, il est possible de garantir deux de ces propriétés à la fois parmi les trois. Par conséquent, la plupart des systèmes de stockage distribués relâchent la propriété de cohérence afin de garantir une meilleure disponibilité et une meilleure tolérance aux pannes.

Plus récemment, une nouvelle conjecture, appelée PACELC, qui s'exprime ainsi : s'il existe une partition (P), comment le système fait un compromis entre disponibilité (A) et Cohérence (C) sinon (E) quand il fonctionne normalement sans partitionnement, comment le compromis s'effectue entre latence (L) et cohérence (C) a été énoncée par D. Abadi [Aba12]. Cette reformulation limite la portée du théorème CAP. Il introduit des objectifs différents si le système fonctionne en mode panne ou non. Le choix de deux des trois propriétés CAP, n'est finalement pas un choix binaire mais un choix à effectuer graduellement en fonction des conditions dans lesquelles le système se trouve. Un nouvel objectif apparaît en mode normal : la latence. La latence n'est pas que la latence réseau, c'est la latence d'un accès à une donnée pour un programme client.

Les bases de données distribuées modernes telles que Dynamo [DHJ+07a], Cassandra [LM10], ou Riak [Klo10] n'assurent pas une cohérence forte par défaut et se fondent sur une cohérence éventuelle. L'objectif est de favoriser les temps de réponse courts, la disponibilité des données, et le passage à l'échelle du système, mais cela se paie avec un risque de lecture de données obsolètes. L'utilisation de politiques de réplication par quorum [Vuk10] permet de limiter ces problèmes et régler finement le niveau de cohérence pour chaque requête ou chaque table de la base [dd15]. Cette approche est appelée cohérence adaptative, et permet à l'utilisateur de choisir exactement le niveau de cohérence attendu pour chaque requête.

L'estimation du niveau de cohérence nécessaire à une requête reste difficile pour le développeur. Certaines approches poussent le concept d'adaptation du niveau de cohérence plus loin en tentant l'adaptation dynamique et autonome en cours d'exécution. Le théorème CAP a ouvert une voie en identifiant des choix d'architecture, PACELC a affiné l'espace des considérations. Ces travaux montrent clairement que le choix entre cohérence forte ou faible n'est pas binaire mais qu'un ensemble de choix intermédiaires existe. L'exploration de ces différents niveaux d'échange permet d'observer les différentes marges de manoeuvre possibles pour le développeur.

Le travail présenté dans cette thèse présente trois approches de gestion de la cohérence des données pour les bases de données distribuées. Dans le reste de ce chapitre, nous résumons les contributions de ces travaux en quatre parties. Nous commenceront dans la partie suivante par un travail sur la simulation des problèmes de cohérence dans les systèmes distribués, qui a permis la simplification de l'évaluation des protocoles mis au point dans cette thèse. Dans la seconde partie de ce chapitre nous décrivons le protocole LibRe, et son implantation dans le système de bases de données distribué Cassandra [LM10]. La description d'un protocole de gestion adaptative de la cohérence fondé sur l'analyse temporelle des requêtes utilisateurs est présentée ensuite. Enfin, la dernière contribution de ce travail est une structure de donnée distribuée, nommée "priority register" (registre a priorité) qui permet au développeur de résoudre les conflits de cohérence par la spécification d'un ordre de priorité sur les données à conserver. Cette section sera suivie par nos conclusions et perspectives sur les travaux menés durant ces trois ans.

Partie 1 - Évaluation de protocoles de cohérence par simulation

Les travaux de cette thèse portant sur l'évaluation et la comparaison de protocoles de maintien de la cohérence, il a été nécessaire de développer un simulateur permettant de comparer et d'exécuter rapidement des tests à grande échelle sur ces protocoles. Cette approche a permis de valider les implantations réalisées en amont de leur déploiement sur une vraie plate forme. Ce travail a été publié dans [LKC14].

Travaux similaires

Les simulateurs de stockage distribués ont d'abord émergés avec l'apparition des grilles de calcul. Ces outils ont ensuite été adaptés pour les architectures de type cloud.

CloudSim [CRB+11b], est une des bibliothèques de simulation les plus populaires, permet de simuler un grand nombre d'applications différentes. Cependant, les métriques fournies par ce simulateur sont généralement d'un niveau de granularité trop grand. Il s'agit par exemple de consommation électrique totale d'un centre de calcul, ou de l'efficacité des machines pour un niveau de service donné, etc.. Cet outil n'était donc pas adapté aux objectifs de la thèse.

OptorSim [BCC+03], qui a été développé dans le cadre du projet européen DataGrid [1] adresse bien la problématique de la répartition mais se concentre plutôt sur le problème de l'évaluation du coût d'un protocole au niveau réseau, au lieu de mesurer son efficacité en termes de latence ou de cohérence pour les clients.

La bibliothèque SimGrid [BLM+12] a été originellement conçue pour simuler les systèmes distribués massifs. Au moment où les travaux de cette thèse ont commencé, le système ne prenait pas encore en compte la simulation des disques ou la simulation des clouds, contrairement au simulateur Simizer que nous avons développé.

Le simulateur GreenCloud[KBAK10], reposant sur le simulateur réseaux *NS-2* [MFF], se concentre sur la modélisation de consommation électrique des centres de données. Cette approche n'était pas adaptée à notre besoin et était peu extensible. Enfin, le simulateur ICanCloud [NVPC+12] se concentre quant à lui sur l'estimation des coûts d'utilisation d'une plate-forme de Cloud cible, comme la plate-forme Amazon EC2 [2].

De part la nature partagée et multi-tenant des infrastructures en nuage, l'évaluation de la performance des applications en mode cloud est un problème difficile [BS10; IOY+11]. Par conséquent, il existe un besoin de fournir des outils de simulation adéquats. Sakellari et al. [SL13a], notait qu'en 2013 aucun simulateur de cloud connu ne permettait d'estimer le comportement d'un application existante déployée dans le cloud. Le peu d'efforts qui

1. `http://eu-datagrid.web.cern.ch/eu-datagrid/`
2. `http://aws.amazon.com`, October, 5th, 2013

ont été faits en ce sens tels que le simulateur CDOSim [FFH12], ne sont pas disponibles publiquement.

Nous avons finalement constaté que les simulateurs de plate-formes Cloud existants n'étaient pas utilisables pour l'étude d'impacts à grande échelle au niveau du centre de données, ou bien qu'ils ne possédaient pas les fonctionnalités nécessaires à la simulation détaillée d'algorithmes distribués, ce dont nous avions besoin.

Nous avons donc développé le simulateur Simizer dans le cadre de cette thèse dans le but de fournir un interface de programmation simple pour modéliser les système distribués et les systèmes de stockage distribués.

Simulation de protocoles de cohérence avec Simizer

SImizer est un simulateur libre à événements discrets écrit en JAVA. Il est basé sur une architecture à trois couches: la couche Application, la couche Architecture et la couche Événements (Event). La couche Événements fournit les classes permettant la gestion des événements et la génération de nombres aléatoires selon différentes lois mathématiques. La couche Architecture fournit les classes permettant de décrire l'environnement de simulation comme les machines, les réseaux... Enfin, la couche application permet à l'utilisateur de décrire ses protocoles et leurs comportements. La couche Application permet aussi de décrire le comportement des clients d'un système. C'est à travers ces interfaces qu'il est possible de décrire une politique de gestion de la cohérence.

Le théorème CAP indique qu'à partir d'une certaine échelle, la cohérence dans un système distribué ne peut être maintenue qu'au prix du sacrifice de disponibilité ou de la tolérance aux partitionnements. Des travaux supplémentaires sur ce sujet ont montré que que la disponibilité du système est liée à la latence des requêtes [Aba12]. Simizer permet donc de comparer les protocoles choisis selon deux métriques principales: le nombre de lectures erronées et le temps d'exécution des requêtes en écriture ou en lecture. Nous définissons une lecture erronée comme la lecture d'une donnée qui n'est pas à jour par rapport au nombre de requêtes en écriture effectuées jusqu'au moment de la requête en lecture.

Le processus de réplication des données est central dans un protocole de gestion de la cohérence. La mise en oeuvre d'un nouveau protocole dans Simizer utilise ce fait en attribuant à chaque donnée (appelée Resource) du système simulé un temps de *disponibilité* (*alive time*). Le temps de disponibilité est le temps au delà duquel une version d'une donnée devient visible à la machine qui la stocke. La politique de gestion de la cohérence est implantée de sorte à calculer le temps de disponibilité pour chaque donnée en fonction des requêtes effectuées par les clients (lectures ou écritures). Une lecture erronée est donc détectée lorsque, lors d'une requête en lecture, la donnée cible n'est pas disponible ou bien qu'elle n'est pas encore visible par le système. Dans le second cas, cela signifie que la donnée n'est pas à jour par rapport à ses répliques.

La gestion de la réplication dans Simizer, se fait en dérivant la classe Processor qui fournit trois méthodes : read, write et update, prenant chacune en paramètre une donnée cible ainsi que sa taille. Chaque politique de cohérence est donc implémentée dans sa propre classe et le choix de la classe à utiliser est un paramètre de la simulation. Cette approche a permis la comparaison de plusieurs politiques de répartition et a donné des résultats acceptables par rapport à des tests dans le cloud.

Résumé et conclusions

Dans cette partie, nous avons résumé le travail réalisé au cours de cette thèse en matière de simulation de protocoles de gestion de la cohérence des données. Dans la thèse nous avons montré que les simulateurs existant à l'époque de ce travail, ne fournissaient pas les fonctionnalités nécessaires ou le niveau d'abstraction approprié pour simuler des protocoles de cohérence. Simizer permet la simulation des protocoles applicatifs eux mêmes, et ainsi, la comparaison de différents protocoles par rapport aux mêmes comportements de clients et à des métriques bien définies. Grâce à Simizer il a été possible de simuler efficacement plusieurs protocoles dans différentes conditions et en amont d'un déploiement dans un système réel. Simizer est une librairie de simulation en logiciel libre, open-source, qui permet une simulation rapide et efficace de systèmes distribués. L'interface de Simizer est intuitive et permet de décrire rapidement une infrastructure et le comportement des clients de l'application simulée. Le développeur / chercheur n'a ainsi plus qu'à se concentrer sur

le développement de son modèle de simulation. A cet effet, cet outil a été particulièrement utile au cours de cette thèse pour le développement du protocole LibRe, décrit dans la section suivante.

Partie 2 - LibRe: un autre rapport Cohérence / Latence pour les bases de données distribuées

La stratégie de réplication affecte le comportement des systèmes distribués en ce qui concerne les propriétés de cohérence, disponibilité et tolérance au partitionnement. Nous savions d'après le théorème du CAP (Consistency, Availability, Partition Tolerance)[GL02a] qu'un système distribué ne peut respecter les trois propriétés à la fois. Ce théorème énonce que tout système distribué peut répondre à deux propriétés parmi trois : la cohérence (Consistency, i.e tous les noeuds du système voient exactement les mêmes données au même moment), la disponibilité (Availability, i.e les données sont toujours accessibles même en cas de panne), tolérance aux partitionnements (Partitionning, l'application continue à rendre le service attendu malgré le partitionnement du réseau). Dans notre étude, nous avons supposé la propriété P toujours satisfaite, il reste alors à faire un choix entre l'une des propriétés C ou A. Par conséquent, les hypothèses traditionnelles, telles que la réplication complète ou le support des transactions, doivent être assouplies. Les solutions existantes diffèrent quant au degré de cohérence des données qu'elles fournissent, un compromis reste à déterminer entre la latence, la cohérence et la disponibilité des données [Aba12].

LibRe, la contribution initiale de cette thèse, est un protocole de gestion de la cohérence permettant de fournir la dernière mise à jour d'une donnée à l'utilisateur tout en ne consultant qu'un minimum de répliques des données. Ceci se fait en stockant la liste des mises à jour estampillées dans un annuaire. Ce protocole a été évalué par simulation grâce à la plate-forme Simizer [LKC14]. Nous avons également évalué LibRe en le mettant en oeuvre au sein du système de stockage Cassandra [LM10] afin de le comparer avec les niveaux de cohérence offerts par ce système.

La suite de cette partie est organisée comme suit : la section suivant décrit l'état de l'art. Dans la section "Description de LibRe", nous présentons notre approche de gestion de la cohérence : LibRe. Enfin, la section "Résumé et conclusion" conclut cette partie.

Le problème de la cohérence des données se pose quand une mise à jour pour une des répliques se produit. Par conséquent, la cohérence des données est principalement influencée par la stratégie de réplication de données adoptée par le système. Daniel Abadi explique dans [Aba12] que les mises à jour se propagent de trois façons différentes selon la stratégie de réplication adoptée (une combinaison de ces modes de propagation peut être envisagée): Synchrone, asynchrone ou hybride.

Synchrone Si les mises à jour sont appliquées de manière synchrone, la lecture des données de n'importe quelle réplique est correcte (cohérence forte). Cependant, le temps de propagation peut être affecté par la lenteur d'un noeud ou son emplacement physique dans le système. Il est d'usage de verrouiller l'accès à une donnée le temps que les mises à jour se propagent afin d'assurer une cohérence forte [SS05]. Dans ce cas, la demande de lecture doit attendre ou doit être abandonnée jusqu'à ce que les données soient disponibles après mise à jour. Par conséquent, cela affecte la disponibilité du système.

Asynchrone Dans l'approche asynchrone, le noeud qui reçoit la première mise à jour l'applique localement et renvoie un message de succès au client. La mise à jour est ensuite propagée à ses répliques de manière asynchrone (en arrière-plan). Dans ce cas, il existe un compromis entre cohérence et latence en fonction de la façon dont les demandes de lecture sont traitées par le système.

— Cas 1: Les requêtes de lecture sont servies à partir d'un noeud particulier en tant que point d'entrée central. Il n'y aura donc pas de sacrifice de la cohérence. Cependant, puisque les demandes sont servies à partir d'un noeud central, il existe un risque de surcharge de ce noeud ce qui peut affecter le temps de latence.

— Cas 2: Si il n'y a pas de point d'entrée central pour les requêtes de lecture, il y a un risque de lecture d'une réplique où la mise à jour récente n'est pas encore appliquée. Dans ce cas, la latence pour la lecture serait minimisée mais en sacrifiant la cohérence des données.

Hybride La combinaison des modes synchrone et asynchrone dans l'objectif d'atteindre un meilleur compromis entre la latence, cohérence et disponibilité est également possible. Ce mode est connu sous le nom de cohérence au vote majoritaire (Quorum consistency). La majorité des répliques forme un quorum. La taille du quorum peut être estimée par la formule $N/2 + 1$, où N est le nombre de répliques. Le système va propager la mise à jour de façon synchrone au quorum et de manière asynchrone pour le reste des répliques. Dans ce cas, la cohérence du système peut être assurée par la formule $W + R > N$ où N est le nombre de répliques, W et R sont respectivement le nombre de répliques contactées pour l'écriture et la lecture.

Par ailleurs, il existe plusieurs niveaux de cohérence qui peuvent être classés de la façon suivante:

Cohérence forte Les systèmes de cohérence forte garantissent que toute lecture sur un élément puisse accéder à la dernière écriture faite sur cet élément. Les systèmes assurant une cohérence forte suivent les principes de sérialisabilité et linéarisabilité afin d'éviter la divergence des répliques [HW90].

Cohérence à terme Les systèmes qui respectent ce type de cohérence, garantissent qu'une mise à jour se propage vers toutes les répliques pour former un état cohérent si aucune autre mise à jour n'a été opérée entre temps. La lecture/écriture est considérée comme réussie si au moins une des répliques répond correctement à l'émetteur de la requête (valide la mise à jour). Les mises à jour sont ensuite propagées à travers le réseau et en arrière-plan de manière asynchrone. Dans ce type de cohérence, il n' y a aucun ordre imposé aux opérations, ainsi il peut y avoir des conflits de mises à jour. Ces conflits sont généralement résolus au niveau du client ou par des règles applicatives [Vog09].

Cohérence causale Ce type de cohérence permet de limiter les conflits qui peut subvenir dans le cas d'une cohérence à terme (comme vu précédemment). Il est également conçu pour des systèmes concurrents multi-lectures/écritures donnant la possibilité à toute réplique de répondre aux requêtes utilisateurs. Toutefois, le système qui vérifie la cohérence

causale impose qu'une mise à jour ne s'opère que si d'autres mises à jour ont été exécutées avant, ainsi les conflits peuvent être évités. En d'autres termes, la pré-condition définit un ordre partiel des opérations qui doivent être exécutées sur la base de leur causalité. Il peut y avoir tout de même des conflits si deux mises à jour sur la même donnée arrivent en même temps. La phase de résolution de conflit est toujours nécessaire. Même si la cohérence causale est plus "forte" que la cohérence à terme, il ne faut pas négliger sa consommation en débit4 et en nombre de messages transmis à travers le réseau.

Cohérence faible Ce type de cohérence est généralement utilisé dans des systèmes hors-ligne. Les répliques ne sont pas tout le temps connectées entre elles, elles peuvent l'être à de courtes périodes qui ne sont a priori pas connues. Quand deux noeuds se connectent, elles partagent leurs mises à jour pour converger vers un état cohérent. Ainsi, si une donnée est modifiée, cette modification sera propagée aux répliques au bout d'un certain temps[SS05]. A noter que d'autres définitions existent dans la littérature [VV15]

Description de LibRe

La principale composante de LibRe est un annuaire qui répertorie toutes les écritures et mises à jour opérées dans le système. Ainsi, un noeud peut être cohérent pour quelques données mais erroné pour d'autres. A titre d'exemple, supposons une topologie où les serveurs sont connectés à un réseau commun. Si un noeud particulier dans la topologie est en panne ou séparé du réseau pendant une période de temps, le noeud sera incohérent pour les opérations qui ont eu lieu pendant cette période. Toutefois, le noeud sera cohérent pour les données non affectées par ces opérations. En d'autres termes, nous souhaitons identifier les mises à jour qui ne se sont pas encore exécutées sur les noeuds. Ceci permet au système d'arrêter de transmettre les demandes au noeud erroné jusqu'à ce qu'il soit à nouveau cohérent. Un noeud est considéré comme erroné s'il contient des données périmées pour la requête entrante. Cette restriction peut être libérée une fois que les mises à jour sont opérées sur le noeud.

Pour cela, les noeuds annoncent leurs mises à jour à l'annuaire. L'annonce contient l'identifiant de la ressource modifiée ainsi que l'identifiant du noeud qui a reçu la mise

à jour. L'annuaire a été implémenté de façon décentralisée selon une table de hachage distribuée (DHT) [ZWXY13] afin d'éviter un unique point d'entrée SPOF (Single Point of Failure) si celui-ci tombe en panne. A noter également que la fiabilité de LibRe, son coût d'exécution ainsi qu'une étude formelle de son comportement ont été effectués. Cette étude ne sera pas présentée dans ici, mais pour plus d'information, le lecteur peut se référer à [KLCG15].

La figure 1 montre où se positionne LibRe dans le système complet.

Nous appelons le "Frontend" le noeud par lequel un client se connecte pour envoyer ses requêtes. Nous considérons une architecture multi-lecture/écriture où chaque noeud peut jouer le rôle du frontend. Le frontend interroge le registre LibRe afin de trouver le noeud cible pouvant répondre à la requête.

Tel que le montre la figure 1, le protocole LibRe est constitué de 3 composants à savoir: l'annuaire également appelé registre (*Registry*), le gestionnaire de disponibilité (*Availability Manager*), le gestionnaire de notification (*Advertisement Manager*). L'annuaire est une structure de stockage en mémoire sous format clé-valeur: la clé étant l'identifiant d'une ressource (donnée) et la valeur est une liste des répliques contenant les données les plus récentes. Si la taille de la liste des noeuds mis à jour atteint le nombre de répliques dans le système, cela signifie que cette donnée est cohérente. L'enregistrement clé-valeur pour cette donnée peut donc être supprimé de l'annuaire en toute sécurité. Le gestionnaire de disponibilité est en charge de transmettre la requête de lecture vers le noeud contenant la version récente des données. Le gestionnaire de notifications quant à lui se charge d'enregistrer dans l'annuaire les notifications des noeuds ayant reçu des mises à jour pendant les opérations d'écritures. Ces composants sont stockés dans les noeuds répliques suivant une architecture de DHT [ZWXY13].

A noter que tous les noeuds contiennent un annuaire. Toutefois, celui-ci n'est pas répliqué, il est co-localisé avec les données dont il est en charge (supervision de leurs versions). En cas de défaillance d'un noeud, son annuaire est reconstruit dynamiquement avec le flot des requêtes transmis aux noeuds responsables de la réplication. LibRe est destiné à être adopté dans des systèmes éventuellement cohérents. Par conséquent la mise à jour de l'annuaire peut ne pas être exactement la même que celle des noeuds de réplication

sans nuire de manière importante à l'utilisateur.

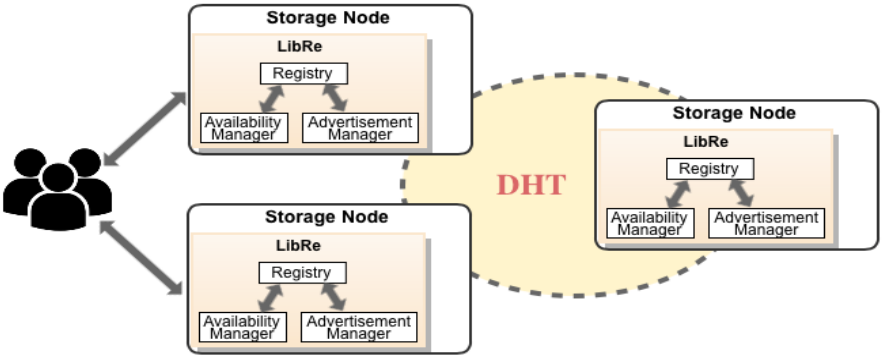
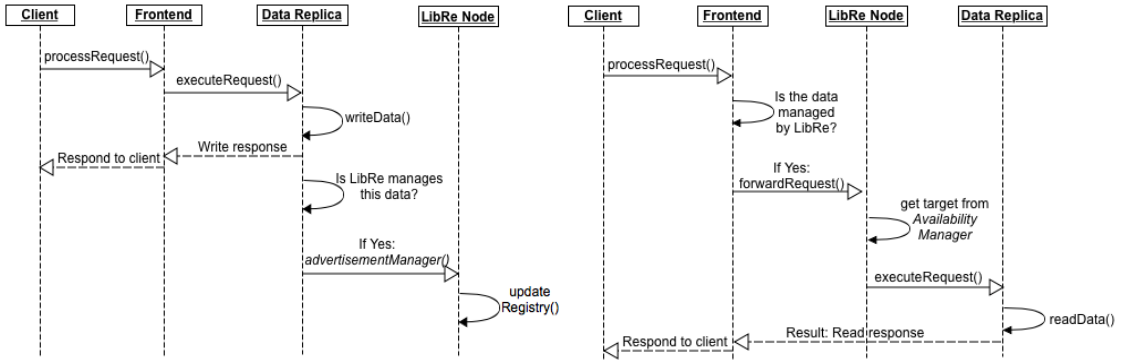


FIGURE 1 – Architecture globale de LibRe



(a) LibRe Write/Update Operation (b) LibRe Read Operation

FIGURE 2 – Diagramme de séquence LibRe

Les figures 2a et 2b montrent le diagramme de séquence du comportement de LibRe pendant les opérations d'écriture et de lecture respectivement.

Opération d'écriture

Dans tout système distribué, quand un frontend reçoit une requête d'écriture, il la transfère à toutes les répliques. Si tous ces derniers lui accusent la bonne réception de la modification, le frontend émet une réponse de succès au client. Si le nombre suffisant de réponses des répliques n'a pas été reçu dans les délais, le frontend émet une réponse d'échec. Le protocole LibRe suit ce même comportement usuel, en l'étendant avec un message de notification que la réplique envoie de manière asynchrone à l'annuaire. Cette notification

est constituée de la clé de la donnée, sa version *version-id* et le noeud ayant reçu la mise à jour.

L'algorithme 3 décrit la contribution du gestionnaire de notifications lors d'une opération d'écriture. On distingue deux cas:

— Insertion: Lorsqu'une donnée est écrite dans le système de stockage pour la première fois ;

— Mise à jour: Quand la donnée à mettre à jour existe déjà dans le système de stockage. La mise à jour peut provenir du frontend ou transférée par un noeud réplique.

Algorithm 1 Opération d'écriture avec LibRe

```
1: function LOG(dataKey, versionId, nodeIP)
2:     if Reg.exists(dataKey) then
3:         RegVersionId ← Reg.getVersionId(dataKey)
4:         if versionId = RegVersionId then
5:             replicas ← Reg.getReplicas (dataKey)
6:             replicas ← appendEP(replicas, nodeIP)
7:             Reg.updateReplicas(dataKey, replicas)
8:         else if versionId > RegVersionId then
9:             replicas ← reinitialize(nodeIP)
10:            Reg.updateReplicas(dataKey, replicas)
11:            Reg.updateVersionId(versionId)
12:        end if
13:    else
14:        replicas ← nodeIP
15:        Reg.createEntry(dataKey, replicas)
16:        Reg.updateVersionId(versionId)
17:    end if
18: end function
```

Quand un noeud réplique envoie un message de notification concernant une mise à jour, le *Gestionnaire de disponibilité* suit les actions suivantes. D'abord il vérifie si la clé de la donnée *data-key* existe déjà dans l'annuaire: la ligne 2. Si oui (opération de mise à jour), ligne 3: la *version-id* pour cette ressource est récupérée. *Version-id* est un nombre (qui croit de façon monotone) représentant le caractère récent de la mise à jour, cela peut être par exemple le timestamp de l'opération.

Ligne 4: Si la *version-id* enregistrée dans l'annuaire correspond à la version-id de l'opération (la mise à jour), alors la ligne 7: l'adresse IP du noeud sera ajoutée avec la liste des répliquas existant. Ligne 8: Si les deux versions-ID ne correspondent pas et si la version-id de l'opération est supérieure à la *version-id* existant dans l'annuaire (ce qui signifie, la mise à jour est nouvelle), la ligne 9 -10: la liste des répliques pour la *data-key* sera réinitialisée à l'adresse IP du noeud. La *Version-id* sera également mise à jour (Ligne 11).

Si *data-key* n'existe pas dans l'annuaire (ligne 13), ce qui signifie que c'est une insertion,

alors une nouvelle entrée sera créée avec la nouvelle *data-key*, l'adresse IP du noeud et la *version-id* de l'opération (lignes 14 à 16). Cette approche correspond à la politique LWW (Last Writer Wins: dernière mise à jour gagnante) [SS05].

Opération de lecture

Quand le frontend reçoit une requête de lecture, il envoie un message de demande de disponibilité (avec la requête et la *data-key*) au noeud hébergeant l'annuaire. Celui-ci envoie la requête aux répliques se trouvant dans l'annuaire. Si aucune entrée n'est trouvée, alors la requête sera envoyée à n'importe quel noeud réplique (cf. figure 2b). L'algorithme 4 décrit ce comportement.

Algorithm 2 Opération de lecture avec LibRe

```
1: function getTargetNode(dataKey)
2:     replicas ← Reg.getReplicas(dataKey)
3:     targetNode ← getTargetNode(replicas)
4:     if targetNode is NULL then
5:         targetNode ← useDefaultMethod(dataKey)
6:     end if
7:     forwardRequestTo(targetNode)
8: end function
```

Résumé et conclusion

Nous avons présenté dans cette section des méthodes de gestion de la cohérence dans les systèmes de stockage de données à large échelle. L'objectif de cette étude est de trouver le meilleur compromis entre la cohérence, la latence et la disponibilité dans ces systèmes. Après étude des différents types de cohérence existants et des algorithmes mis en oeuvre dans les systèmes actuels, nous nous sommes rendus compte qu'il existait une possibilité non encore explorée: assurer la cohérence la plus forte possible tout en ne consultant qu'une seule réplique. Pour cela, nous avons proposé un algorithme appelé LibRe que nous avons évalué par simulation grâce à la plateforme Simizer mais aussi dans un cas réel en l'intégrant au système de stockage Cassandra. Les résultats de ces expérimentations ont démontré l'efficacité de notre approche.

Partie 3 - Cohérence adaptative

Une des limitations majeure des systèmes de gestion de données distribuées modernes est l'inévitable intermédiation entre cohérence, disponibilité et latence des requêtes issue du théorème de CAP. Un contournement de ce problème peut être trouvé en concevant le système de sorte qu'il choisissent automatiquement le niveau de cohérence nécessaire au bon fonctionnement du système en fonction de l'application ou du contexte. Cette fonctionnalité s'appelle la "cohérence adaptative". Grâce à la cohérence adaptative, un développeur peut optimiser précisément la performance de son application en traitant différemment les données ou les requêtes nécessitant un fort niveau de cohérence et celles qui n'en ont pas besoin. Cependant, ces décisions dépendant souvent d'invariants applicatifs ou de phénomènes externes, décider du niveau de cohérence automatiquement reste un problème ouvert, surtout dans le cas de systèmes où le comportement des utilisateurs change au cours du temps.

travaux similaires

La littérature existante sépare les travaux en matière de cohérence adaptative en deux grandes catégories: définie par l'utilisateur, ou définie par le système.

Définie par l'utilisateur : Ces systèmes permettent au développeur, grâce à un ensemble de paramètres, de définir le niveau de cohérence attendu pour chaque opération sur le système. Ces opérations peuvent être la création, la modification, la lecture ou la suppression d'une donnée. Le développeur décide donc soit statiquement, soit dynamiquement de choisir un niveau de cohérence donné pour une requête. Cette souplesse de configuration des requêtes est un modèle sûr car le développeur de l'application est en général le mieux placé pour connaître le comportement du système et donc choisir un niveau de cohérence approprié. La plupart des systèmes de bases de données distribués modernes tels que Cassandra [LM10], Amazon Dynamo [DHJ+07a] ou Riak [Klo10] offrent cette possibilité. Dautres travaux dans ce domaine sont les travaux sur la Red-Blue consistency [LPC+12] et SALT[XSK+14].

Définie par le système Dans ce modèle de cohérence adaptative, le système de gestion de données sélectionne automatiquement le niveau de cohérence à attribuer à chaque requête. Le principal défi de cette approche consiste à catégoriser correctement les requêtes dans un niveau de cohérence avant leur exécution. A cet effet, différentes approches peuvent être utilisées, telles que la liste des opérations précédentes, la valeur de la donnée elle-même, la probabilité d'apparition d'un conflit, la charge réseau, la latence, etc... Différentes approches existent dans la littérature comme la cohérence continue [YV00], IDEA [LLJ07] ou Harmony [CIAP12]. Certaines méthodes, comme le rationnement de cohérence [KHAK09] utilisent un mélange de ces deux approches.

La plupart des approches "systèmes" utilisent un service séparé au-dessus du système de stockage pour ajouter le paramètre approprié à la requête avant son exécution. Ce type d'architecture peut donc être aisément sujet à un goulot d'étranglement. Nous pensons que l'intégration de ces techniques au niveau du système de stockage lui-même permet de réduire considérablement la surcharge provoquée par l'analyse des requêtes. De plus, cette approche permettrait de régler plus finement les choix du système sur la base d'une requête ou d'une session plutôt qu'au niveau d'une table entière.

Changement dynamique du niveau de cohérence

L'approche développée dans cette thèse consiste à mettre à jour dynamiquement le niveau de cohérence des requête sur la base de données, à partir d'un niveau de cohérence par défaut. Ce niveau par défaut est le plus bas, dans lequel un seul noeud est écrit ou lu à chaque requête. A partir de l'analyse temporelle des requêtes effectuées sur le système, l'utilisateur spécifie des plages de temps et de données avec des niveaux de cohérence différents du niveau par défaut. Ces informations permettront au système de détecter rapidement si les requêtes en cours on besoin d'utiliser un niveau de cohérence plus fort.

L'information sur les données nécessitant un niveau de cohérence plus fort est diffusée à tous les noeuds coordinateurs du système. Comme ce type de noeud est très inférieur au nombre total de noeuds dans le système, cela ne représente pas une grande quantité de données. Les données fournies aux coordinateurs consistent en un ensemble de *clefs* identifiant des données stockées dans le système. A chaque clé est adjointe un niveau de

cohérence et un intervalle de temps avec une heure de départ et une heure de fin. A la réception de ces données, les identifiants ayant des intervalles et un niveau de cohérence similaires sont regroupés dans un filtre de Bloom [BMM02]. Ceci permet de compresser largement l'information contenue dans les listes.

À la date correspondant au début de l'intervalle associé à un filtre, le filtre est basculé en mode "actif" sur tous les noeuds coordinateurs. Lors d'une opération en lecture ou en écriture, le noeud va vérifier la présence de la donnée cible dans les filtres actuellement actifs. Si la donnée est présente dans l'un des filtres, alors le système applique le niveau de cohérence correspondant à ce filtre. Si l'identifiant est introuvable dans les filtres, alors le niveau de cohérence par défaut sera appliqué.

Le filtre de Bloom étant une structure de données probabiliste, il existe un risque de faux positif, indiquant l'élément est présent dans le filtre alors qu'il n'en fait pas partie en réalité. Dans le cadre de notre approche, cela implique qu'un élément présent dans une liste pourrait se voir appliquer un niveau de cohérence différent de celui attendu par l'utilisateur. C'est pour cette raison que le niveau de cohérence le plus bas a été choisi par défaut: en cas de faux positif, un élément se verra toujours appliquer un niveau de cohérence plus élevé que celui qui lui a été attribué. Cet approche conservatrice permet de limiter les risques de lectures erronées, au détriment de la performance pour une petite partie des données.

La figure 3, schématise ce fonctionnement : lors de l'arrivée d'une requête sur le noeud Coordinateur, notre algorithme de sélection de cohérence itère sur les filtres de Bloom présents sur le coordinateur, jusqu'à trouver l'identifiant utilisé dans la requête reçue.

Ce modèle a été développé et déployé dans une version modifiée du système de stockage Cassandra, et a été testé sur un cas d'application représentatif. Ce cas d'application est fondé sur les besoins du système Vélib', le système de partage de vélos parisien. Ce système nécessite l'application d'un niveau de cohérence fort pour fournir l'information sur la disponibilité des vélos dans ses stations. Or, il est plus important de garantir la précision de cette information lorsque le nombre de vélos utilisables ou que le nombre d'emplacements libres sont très bas. Ainsi, grâce aux travaux sur le partitionnement de séries temporelles décrits par Chabchoud et al. dans [CF14], il a été possible de créer une série de filtres

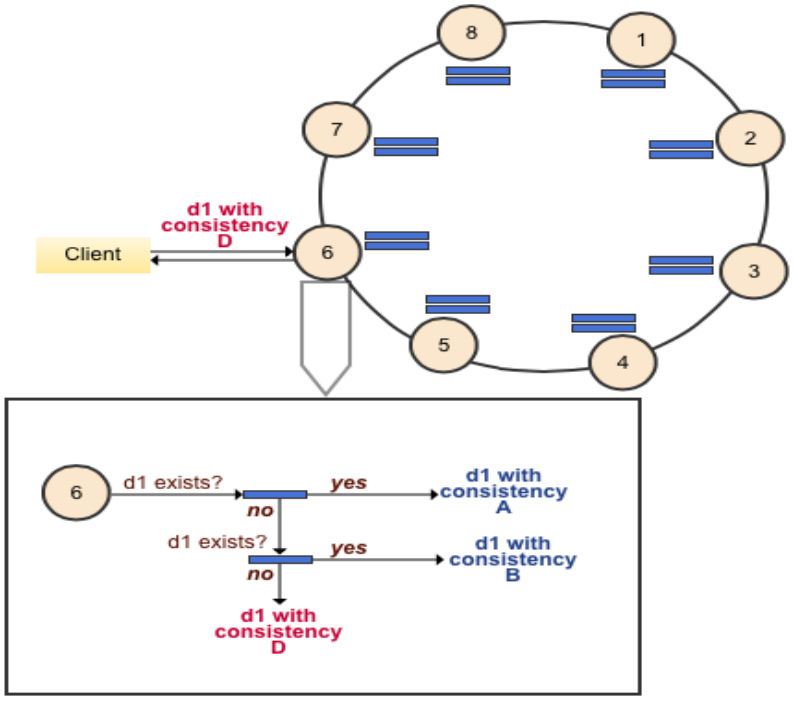


FIGURE 3 – Overriding Application-defined Consistency options

permettant d'ajouter un niveau de cohérence plus fort pour les stations les plus demandées. Nous avons testé notre approche en rejouant les requêtes du système à un rythme plus élevé pour créer des lectures erronées.

Résumé et conclusions

Les travaux décrits dans cette section proposent une nouvelle méthode de cohérence adaptative définie par le système. Notre méthode est intégrée directement au sein du système de stockage, pour limiter la surcharge impliquée par l'analyse des requêtes pour la sélection du bon niveau de cohérence. Nous avons utilisé l'analyse de séries temporelles pour repérer les données qui ont besoin de niveaux de cohérence différents du niveau standard. Ces éléments sont regroupés dans des listes compressées grâce aux filtres de Bloom et le système utilise ces filtres pour déterminer dynamiquement le niveau de cohérence requis pour chaque requête. Nous avons testé cette approche sur le système Cassandra et les résultats sont encourageants.

Partie 4 - Registre de priorité: Ordre de remplacement spécifique pour la réconciliation

La dernière contribution de cette thèse se concentre sur les problématiques de résolutions de conflits entre répliquas. En effet, les systèmes de stockage récents troquent la cohérence des données pour un meilleur temps de réponse, et laissent donc de temps en temps les répliquas diverger vers des valeurs différentes. La réconciliation est le processus durant lequel les conflits entre répliquas divergents [Mai08]. Identifier la version la plus à jour entre les différentes versions d'une même donnée et la diffuser le plus rapidement aux répliques est un défi clé de la réconciliation.

Nous décrivons dans cette section un nouveau type de donnée distribué appelé le "registre à priorité". Après avoir introduit quelques notions clés sur la résolution des conflits entre répliquas, nous décrirons la motivation et l'approche développées dans ces travaux.

Notions-clés

En fonction de la technique de détection de conflits, les processus de réconciliation peuvent être classés en deux types: réconciliation syntaxique et réconciliation sémantique [DGMS85; Mai08]. Les techniques de *reconciliation syntaxique* sont fondées sur les relations de sérilisabilité ou de causalités qui peuvent être capturées par les différents types d'horloges logiques: vecteurs d'horloges [Fid88], vecteurs de versions [PPR$^+$83], ou bien encore des horloges physiques. Les techniques de réconciliation syntaxiques sont rapides, efficaces et peuvent être gérées automatiquement par le système. Cependant, elles ne sont pas idéales dans tous les cas. De plus, en combinaison avec des stratégies de réplications optimistes, ces techniques sont sujettes à des problèmes de conflits syntaxiques, qui ne peuvent êtres résolus que par la *reconciliation sémantique*.

La *réconciliation sémantique* est utilisée lors des conflits syntaxiques, ou lorsque le critère de cohérence ne peut être fondé sur les principes de sérialisibilité ou les relations de causalité. Les techniques de réconciliation sémantique sont fondées sur la connaissance spécifique du domaine pour la résolution des conflits entre répliquas. Cette technique peut être appliquée dans un grand nombre de cas d'utilisation. Cependant, ces techniques peuvent

être lentes ou complexes et sujettes aux erreurs [SPBZ11b]. Certains types de données rassemblent les bénéfices des deux approches en une. Ces types sont appelés *conflict-free replicated data types (CRDTs)* [SPBZ11b]. Les CRDTs utilisent la sémantique des opérations disponibles sur le type de données utilisé pour fournir des garanties de cohérence à terme plus ou moins fortes, en fonction du type: registres, compteurs, ensembles, graphes, ou autres [SPBZ11a; Bur14]. Il peut arriver qu'aucune de ces variantes ne soit applicable pour une application cible. Dans ce cas les développeurs utilisent des techniques de réconciliation sémantique pour détecter et résoudre les conflits [RD15b].

Une technique fréquente est l'utilisation de l'action des utilisateurs pour décider de la version à conserver. Un exemple célèbre de cette approche est le cas du panier d'achat utilisé dans les travaux sur le système de bases de données distribué Dynamo[DHJ+07a]. En cas de conflits entre deux versions du panier d'un même utilisateur, le système demande directement à l'acheteur quel panier est le bon. Cette approche de résolution est lente et entraîne une mauvaise expérience utilisateur et ralentit le processus d'achat, ce qui peut nuire à la réputation du site.

Nous pensons donc que d'autres techniques doivent être appliquées pour résoudre ce type de conflits.

Cas d'utilisation et motivation

Ce travail a été motivé par le modèle de *connaissance partagée à ordre partiel* pour les systèmes distribué faiblement couplés [SKT14].

Ce modèle a été utilisé pour les objets de données du projet de réseaux fondé sur le contenu ENCODERS [3] . Dans ce projet, les objets circulant sur le réseau, sont mis en cache afin d'améliorer leur dissémination et leur accessibilité. La technique d'éviction de cache utilisée se fonde sur les méta-données associées à chaque objet. L'utilisateur peut spécifier un ordre de remplacement pour une classe d'objets possédant certains attributs (méta-données). L'ordre utilisé est par défaut un ordre lexicographique sur les attributs (chacun d'entre eux se voit attribué une priorité), et les valeurs associées à ces attributs.

L'ordre global définit sur tous les objets du système est appelé *ordre de remplace-*

3. Edge Networking with Content Oriented Declarative Enhanced Routing and Storage

ment défini par l'application. Cet ordre global n'est que partiel car les objets de classes différentes (qui n'ont pas d'attributs en commun) ne peuvent pas être ordonnés les uns par rapport aux autres. Cette forme de remplacement peut être vue comme une forme de réconciliation sémantique. Par exemple, supposons que le framework ENCODERS ait besoin d'être étendu pour fonctionner dans le Cloud. Une base de données distribuée devrait être utilisée pour stocker les objets du framework et permettre la contribution entre plusieurs réseaux à travers le cloud. Cependant la grande quantité d'objets nécessiterait que la base de données suive le même système de remplacement qu'ENCODERS pour économiser l'espace de stockage.

Par conséquent nous proposons d'étendre une base de données distribuée pour utiliser le même principe de remplacement pour résoudre les conflits entre répliques.

Le registre à priorité

Dans cette section nous décrivons le registre à priorité, qui est un registre paramétré par un ordre défini par l'utilisateur. Ce registre est un registre multi-valué, qui résout les conflits en utilisant un ordre définit sur les valeurs stockées dans le registre au lieu d'utiliser les relations entre opérations. En tant que registre valué, ce registre peut contenir plusieurs valeurs, chacune maximales dans leurs ordres respectifs.

Un cas particulier est celui d'un registre où une seule classe d'objets peut être contenue. Dans ce cas plusieurs objets différents peuvent être stockés si leurs méta-données sont égales. Dans le cadre de la preuve de concept développée au cours de cette thèse, ce type d'égalité est géré en utilisant le timestamp (horodatage) des données pour départager les répliques.

Une application définit un ordre partiel en déclarant pour une classe d'objets, la liste des attributs à utiliser dans l'ordre lexicographique voulu. Notre preuve de concept étant développée sur le système de stockage Cassandra, cela s'est traduit dans notre implémentation par l'introduction de paramètres supplémentaires pour marquer les attributs couvert par l'ordre partiel. En cas de conflit, le système compare les différentes versions à l'aide de l'ordre défini et ne conserve que la version qui est la plus haute dans l'ordre. En cas d'absence de certaines valeurs, les valeurs manquantes sont considérées comme étant

minimales dans leurs valeurs respectives.

Résumé et conclusions

Les techniques de réconciliation syntaxiques ne peuvent pas être appliquées à tous les cas d'usage. L'utilisation de techniques de réconciliation sémantiques s'avère donc nécessaire, car elles peuvent se décliner sur des cas très spécifiques, cependant ces techniques sont parfois lourdes à mettre en oeuvre.

Le registre à priorité devrait permettre de combiner le meilleur de ces deux approches en permettant une grande expressivité au niveau applicatif tout en bénéficiant d'une grande efficacité dans la résolution des conflits. Grâce à ce système les techniques de résolutions coté clients peuvent être intégrées coté serveur à un coût relativement faible.

Partie Conclusion globale

Cette thèse expose trois solutions à des problèmes récurrents de gestion de la cohérence dans les bases de données distribuées. Dans l'ordre chronologique, la première contribution au domaine est la mise au point du protocole LibRe qui permet de fournir un nouvel échange entre cohérence et latence dans les bases de données distribuées, en particulier en cas de faible connectivité.

Un second travail a été effectué dans le domaine de la cohérence adaptative, avec le développement d'une preuve de concept utilisant le partitionement de séries temporelles pour permettre à l'utilisateur d'identifier les intervalles de temps et les données nécessitant un niveau de cohérence plus élevé que la normale.

La contribution principale de cette thèse est la définition du "priority register" qui est un type distribué permettant de paramétrer un ordre de remplacement sur ses valeurs. Cette fonctionnalité permet d'assurer une forme de réconciliation sémantique des réplicas en conflit, sans faire appel à l'utilisateur de l'application.

Chacune de ces contributions a fait l'objet d'une implémentation dans le système de gestion de bases de données distribuées Cassandra.

Ces travaux offrent de nombreuses perspectives de recherche. Tout d'abord sur les

relations entre les proprietes de coherence et les proprietes souhaitees d'un systeme distribue. Pas seulement disponibilité et tolérance aux fautes, mais par exemple prouvabilité ou respect des specifications du systeme. Le protocole LibRe, par sa flexibilité basée sur l'utilisation d'un registre, offre des perspectives interessantes au probleme de la reconciliation du point de vue des applications et plus seulement du point de vue du systeme.

Sur le registre à priorité, son extension à d'autres formes d'ordres mérite d'être étudiée, afin de permettre le support de nouveaux cas d'utilisation, particulièrement dans le traitement des données massives.

Chapter 1

Introduction

The growing amount of information to be stored and processed has increased the importance of efficient data and computational resource management. This phenomenon is expected to grow in amplitude in the near future and is referred to as "Big Data" management. This huge data increase raises great challenges for data management lifecycle and requires innovative approaches to organize, manage, and explore the overwhelming amount of information. The scaling requirements of big data management demands a distributed database system that allows to configure tradeoffs according to the application needs.

In distributed database systems, data replication is inevitable to improve the performance and availability of the system. Data replication is the process of replicating the same data among various storage elements (nodes). Although the physical copy of a data item varies, logically they are one and the same. The number of physical copies and the location of the copies can vary based on the application needs. By introducing more than one physical copies of a data item via replication, an immediate problem that a storage system encounters is Data Consistency. Data Consistency is not violated as long as there are no updates in the system. When a data item is modified on one of the replicas, the same data item stored on other replica nodes becomes stale. The copies of same data item stored on different nodes are called replicas and the nodes storing the replicas are called the replica nodes. Hence updating the same data before any of the replica nodes start processing on the stale replica is very important. Due to this case, a Replica control protocol [LAEA95b],

which can safely handle the CRUD (Create, Read, Update, Delete) operations, forms the core of the replication service. The Replica Control Protocol drives the desirable properties of the storage system that includes Latency, Consistency, Availability, Scalability and Partition Tolerance.

According to CAP conjecture [GL02a], a distributed storage system can not ensure Consistency and Availability at the same time in case of network partition. Due to this reason, most of the systems that prefer to remain highly available need to relax consistency guarantees without affecting the application invariants. Abadi in his paper [Aba12] proposed a new conjecture called PACELC taking into account the request latency and discusses the different tradeoffs between Consistency, Availability and Request Latency. Although, one has to be sacrificed among Consistency and Availability, the sacrifice is not a binary option, it can be tuned to different levels depending on the use case. The consistency guarantees could be sacrificed to a certain limit for the benefit of Latency and Availability. The first level of tradeoff could go between Consistency and Latency, the level of consistency could be sacrificed till certain extent for the benefit of request Latency. In the second level, the level of consistency could be sacrificed further for the benefit of Availability and the third level is a pure decision between the Consistency and Availability.

Modern distributed data storage systems such as Dynamo [DHJ+07a], Cassandra [LM10], Riak [Klo10] do not ensure strong consistency by default, instead, they ensure eventual consistency. Although eventual consistency favors request latency, availability and scalability of the system, they are prone to consistency issues such as Stale Reads and Update Conflicts. In order to better address the consistency issues, these storage systems use quorum-based replication policies [Vuk10] and enables to tune the tradeoff between Consistency, Latency and Availability per query basis. Tuning the tradeoffs involves specifying the number of nodes to contact during read and write operations. The intersection of one or more nodes that are contacted during both read and write operation ensures the consistency of the data, which is also known as *Quorum Intersection Property*. The user/application specifies the number of nodes to contact for each request and the system handles the requests with the appropriate consistency level. Some databases like Voldemort [SKG+12], use quorum replication policy and define the consistency level for each

table in order to tune the tradeoff per table basis instead of each query [dd15]. This practice of using the appropriate consistency level based on the application/user needs or depending on the criticality of the requests is called Adaptive Consistency.

Although these systems offer adaptive consistency, in addition to offering higher latency cost for stronger consistency options, it is difficult for the application developers to decide the needed consistency option for a particular request/ data item in advance. In order to acquire the benefits of adaptive consistency feature to a higher extent, adapting the consistency needs of each query dynamically during run time depending on various environmental factors is desired. In addition, the choice between the consistency and availability of these quorum-based replication systems remains binary. If the system could not ensure the liveness of a sufficient number of replica nodes (to ensure intersection property), the system rejects the operation affecting the system availability. In order to finely tune the choice between consistency and availability, various intermediate consistency options between the default eventual consistency and the stronger consistency options that are derived via the quorum intersection property will be helpful.

1.1 Contributions

The contributions of this Ph.D thesis are summarized as follows:

1.1.1 Better Consistency-Latency tradeoff for quorum-based replication systems

As stated by Abadi in [Aba12], request latency works closely with Consistency and Availability of the system. The modern distributed database systems that are based on Dynamo [DHJ+07a] bloodline such as Cassandra [LM10], Riak [Klo10] and Voldemort [SKG+12] sacrifice consistency not only for the favor of availability and partition tolerance, but also for the favor of request latency. These database systems follow quorum-based replication for replica control. Quorum-based replication systems aka NRW systems ensure consistency based on the number of nodes contacted during write and read operations. These systems ensure consistency based on the formula $R + W > N$, where R signifies the number of nodes acknowledged for Read operation, W signifies the number of

nodes acknowledged for Write operation and N signifies the number of replica nodes configured for the particular data item. The formula $R + W > N$, is the intersection property of the quorum, or overlapping quorum.

In order to ensure minimum request latencies, these modern database systems do not satisfy the quorum intersection property by default, and hence they compromise strong consistency guarantees for the favor of request latency. However, in order to ensure strong consistency on demand, the system has to satisfy the quorum intersection property ($R + W > N$) by contacting a sufficient number of replica nodes. One of the limitations in satisfying the quorum-intersection property is that the number of replica nodes to be contacted during read and write operations increases linearly with the number of replica nodes of a particular data item. Hence in order to overcome this limitation, we proposed a new consistency protocol for quorum-based replication systems called LibRe. LibRe is an acronym for **Lib**rary for **Re**plication. As the name suggests, in LibRe, we maintain a library of information about the recent version identifier for each data item and the list of replica nodes that contain the recent version id. By forwarding the read requests to an up-to-date replica node that contains the data with the most recent version-id, the protocol ensures consistency of read operations with minimum read and write latencies. The proof-of-concept of the protocol was implemented inside the Cassandra distributed storage system and was benchmarked against Cassandra's native consistency options. The results of our experimentation confirm that LibRe ensures consistency with minimum read and write latencies.

1.1.2 Evaluation of consistency protocols via simulations

Large-scale system studies are often challenging due to the cost and time frame involved in the deployment of related software and physical infrastructure. In this context, testing a new protocol or strategy could be cumbersome unless relying on appropriate simulation tools. These tools are very useful for evaluating new policies or strategies in a fast and efficient way before testing them in real world. In the literature there are several large-scale distributed systems simulation tools starting from Grid architecture to Cloud architecture. Some of those popular simulation libraries are SimGrid [CLQ08], OptorSim [BCC+03],

GridSim [MB02] and CloudSim [CRB+11a]. Most of these simulation tools are focused on evaluating various features that are related to physical infrastructure and do not facilitate to simulate application or application-level semantics. Since evaluating data consistency is closely related to application semantics and application related constraints, it is difficult to evaluate data consistency with the existing simulation tools. Hence, in order to evaluate data consistency via simulation, we have implemented a cloud-based simulation tool called Simizer [SL13b], which is focused on simulating application level semantics. Simizer is an event-driven simulation tool written in JAVA language. Using Simizer, users can extend the existing application class to override the default application behaviors such as read, write, processing and can implement the needed application level semantics. Simizer takes input about the physical infrastructure, workload pattern and evaluates the application performance and prints the results in an output file. The needed metrics can be studied by processing the output file.

1.1.3 Evaluating Data Consistency on the fly using YCSB

Due to a massive data explosion in the last decade, revolution in storage systems led to the evolvement of modern distributed database systems. These systems are not ACID [Ram03] compliant, instead they rely on BASE (Basic Availability, Soft state, Eventual consistency) paradigm [Pri08]. In order to achieve optimal performance, these storage systems play different tradeoffs depending on the application needs. One of the indispensable among that is the tradeoff between Consistency, Latency and Availability.

Existing database benchmarks such as TPC-Class [1] are not well suited to evaluate the modern data storage systems (at the time of YCSB implementation) due to their query interface and workload patterns. For these reasons, Yahoo! [2] developed a benchmark tool: the Yahoo Cloud Serving Benchmark (YCSB) [CST+10], which intends to benchmark different NoSql systems under similar workload patterns. Although YCSB offers good support for different database interfaces and workload patterns, the evaluation metrics of YCSB do not take into account data consistency of the system. This makes YCSB insufficient to evaluate system tradeoffs between consistency, latency and availability. Hence, in order to

1. http://www.tpc.org
2. www.yahoo.com

enable YCSB to evaluate the system tradeoffs between consistency, latency and availability, we have extended the YCSB code base to add a new evaluation metrics about data consistency. With the extended YCSB code, users can assess the different performance metrics of the system along with the number of stale reads encountered by the test client. The extended YCSB code has been made public for the open source community to take advantage of it [Kum15].

1.1.4 Overriding application-defined consistency option of a query during run-time

As eventual consistency belongs to weaker consistency guarantees, most of the eventually consistent data stores offer additional stronger consistency options and adapt the consistency guarantees of the requests per query basis. These systems enable the user/ developers to specify the needed consistency option for each query and the system processes the request according to the specified consistency option. One of the limitations of this method is the application developers have to decide the needed consistency options for each query in advance during the development time. There are very few works in the literature, that try to adapt the consistency option of the application queries during run-time instead of the application development time.

However, most of the existing works analyze different factors that can influence the consistency needs of the system and adapts the consistency guarantees of the whole system according to the absorbed factors such as the ones described in [CPAB13; MSV+10]. On the other hand, some of the approaches finely tune the adaptive consistency feature on a per query/ data item basis according to the current environment factors. However, in the latter approach, the application queries have to pass through a centralized service in order to query the data with an appropriate consistency option. The limitation of this approach is that the throughput and performance of the centralized service could become a bottleneck for the system. In order to overcome this limitation, we propose to facilitate the modern database systems to adapt the consistency options of the incoming queries according to an external input. The input can be given by the database administrator or by an external monitoring service. By this way, the necessity to pass the application

queries through a centralized service will be avoided and instead, the participation of the external services will be moved to the background for providing input about the consistency decisions. Based on this input, the system can adapt (override) the consistency options of the application queries that are defined during application development time by the needed consistency options.

1.1.5 Application-defined Replacement Orderings for Ad Hoc Data Reconciliation

Reconciliation of replicated data items is one of the major challenges in reaching consistency among the replicated data items. As eventually consistent data stores rely on partial ordering of the updates on the system, the ordering often leads to update conflicts. Data reconciliation is often required to verify the values of conflicting updates on the different replicas and to arrive at a single agreed upon value among all the replicas. Data reconciliation could be broadly classified into two types such as Syntactic Reconciliation and Semantic Reconciliation. Syntactic reconciliation techniques use serializability or causality as the basis for conflict-detection and resolution. These techniques are fast and efficient and the resolutions can be directly managed at the database side. However, for use cases where conflict resolution demands knowledge of the application semantics, domain-specific semantic reconciliation is required. Semantic reconciliation techniques often involve maintaining all possible values and perform the resolution via domain-specific knowledge at the client side. This involves additional costs in terms of network bandwidth and latency, and considerable complexity. In this thesis, we discuss the design of a novel data type called priority register that implements a domain-specific conflict detection and resolution scheme directly at the database side, while leaving open the option of additional reconciliation at the application level. Our approach uses the notion of an application-defined replacement ordering and we show that a data type parameterized by such an order can provide an efficient solution for applications that demand domain-specific conflict resolution. We also describe the proof-of-concept implementation of the priority register inside the Cassandra distributed storage system.

1.2 Organization of the Manuscript

This thesis document is organized as follows:

In the next chapter we provide the state-of-the-art information about the consistency of the replicated data in eventually consistent data stores. This includes information about the evolvement of eventually consistent data stores and its underlying paradigm, different categories of consistency options including its tradeoffs, and the need for adaptive consistency and its challenges.

The contributions of this thesis are focused on two types of inconsistency issues that are normally encountered by the eventually consistent data stores. The first part that includes Chapters 3, 4, 5 and 6 deals with consistency issues related to Read-Read inconsistency (stale reads) and the second part that includes Chapter 7 deals with the issues related to Write-Write inconsistency (update-conflicts).

Chapter 3 describes the design and working principle of a new consistency protocol called LibRe that helps to read the most recent version of a needed data item with minimum read and write latencies. In Chapter 3, we first discuss the general idea of the LibRe protocol and the enhancement of the protocol using Distributed Hash Table (DHT). We also formally describe the behavior and properties of the protocol in Chapter 3. Chapter 4 describes the need and implementation of a new simulation toolkit called Simizer that helps to evaluate different consistency protocols via simulation. We also show the performance evaluation of LibRe against existing consistency protocols using simizer. In Chapter 5, we discuss the prototype implementation of the DHT-based LibRe protocol inside the cassandra distributed database system, which is named CaLibRe: Cassandra with LibRe. The extensions to Yahoo Cloud Serving Benchmark (YCSB) in order evaluate consistency-latency tradeoff and the evaluation results of CaLibRe against Cassandra's native consistency options are also included in the same chapter. Chapter 6 discusses the adaptive consistency feature offered by the modern distributed database systems and its limitations, followed by an approach to address the limitations.

Data Reconciliation is one of the inevitable measures used by the eventually consistent data stores. Chapter 7 provides a simple means to reconcile the conflicting replica states

via a new register data type named Priority Register that reconciles the data items via application-defined replacement orders. The description about application-defined replacement orders and a prototype implementation of the Priority Register inside Cassandra including a case-study evaluation are included in the Chapter 7. Chapter 8 summarizes the contributions of this thesis with concluding remarks and discusses the future dimensions of the thesis work.

1.3 Publications

1.3.1 International Publications

— **AKDM 2016 (Journal):** Publisher: Springer, Sathiya Prabhu Kumar, Sylvain Lefebvre, Raja Chiky, Olivier Hermant, LibRe: A Better Consistency-Latency Tradeoff for Quorum Based Replication Systems, Advances in Knowledge Discovery and Management Vol. 7 (AKDM-7), 2016 (**To Appear**).

— **SCDM 2015:** Publisher: IEEE, Sathiya Prabhu Kumar, Sylvain Lefebvre, Minyoung Kim, Mark-Oliver Stehr, Priority Register: Application-defined Replacement Orderings for Ad Hoc Reconciliation, 3rd Workshop on Scalable Cloud Data Management, 2015, Santa Clara, CA, U.S.

— **GLOBE 2015:** Publisher: Springer, Sathiya Prabhu Kumar, Sylvain Lefebvre, Raja Chiky, Eric-Gressier Soudan, CaLibRe: A Better Consistency-Latency Tradeoff for Quorum Based Replication Systems, International Conference on Data Management in Cloud, Grid and P2P Systems, Valencia, Spain.

— **IWCIM 2014:** Publisher: IEEE, Sathiya Prabhu Kumar, Sylvain Lefebvre, Raja Chiky, Evaluating Consistency on the fly using YCSB, International Workshop on Computational Intelligence for Multimedia Understanding, Prais, France.

— **PaPEC 2014:** Publisher: ACM, Sylvain Lefebvre, Sathiya Prabhu Kumar, Raja Chiky, Simizer: evaluating consistency trade offs through simulation, EuroSys-2014,

Amsterdam, Netherland.

— **ACMCompute 2013:** Publisher: ACM, Sathiya Prabhu Kumar ; Raja Chiky ; Sylvain Lefebvre ; Eric-Gressier Soudan, LibRe: A Consistency protocol for Modern Storage Systems, ACM Compute 2013, Vellore, India.

— **ICSCS 2012:** Publisher: IEEE, Sylvain Lefebvre, Raja Chiky, Sathiya Prabhu Kumar, ISEP, WACA: Workload And Cache Aware Load Balancing policy for web services, 1st International Conference on Systems and Computers Science, 2012.

1.3.2 National Publications

— **EGC 2016:** Raja Chiky, Sathiya Prabhu Kumar, Sylvain Lefebvre and Eric Gressier-Soudan, LibRe: Protocole de gestion de la coherence dans les systemes de stockage distribues.

— **RNTI 2013:** Sylvain Lefebvre and Sathya Prabhu Kumar and Raja Chiky, WACA: Politique de repartition de charge des services web dans une architecture de type Cloud, Revue des Nouvelles Technologies de l'Information, 2013.

— **NOTERE 2012:** Sylvain Lefebvre, Raja Chiky, Sathiya Prabhu Kumar, WACA: Politique de repartition de charge des services web dans une architecture de type Cloud, in Conference annuelles des NOuvelles TEchnologies de la REpartition, 2012.

Chapter 2

State of the Art

Ensuring consistency of replicated data in distributed database systems has always been challenging since past decades. Consistency models ensure the correctness of execution of any distributed program/ application. As stated by Lamport in [Lam89], a distributed program consists of two important properties namely Safety and Liveness. The safety property includes all the rules that ensure the bad things (invariance violation) do not happen. The liveness property guarantees that the system will make progress despite of criticality of the system. The intersection of these two properties ensures the correctness of the distributed program. It is not possible for a distributed program to hold properties that are strong in both safety and liveness. The distributed program has to do some tradeoff between the safety and liveness properties according to the application needs in order to achieve an optimal system. In this chapter, we first introduce the fundamentals of consistency models in Section 2.1 and 2.2. In Section 2.3, we describe some of the fundamental consistency models that ensure different levels of consistency guarantees. The evolution of modern database systems and its underlying principles are discussed in the Section 2.4. The role of replica control protocol and its types are discussed on the Section 2.6. The server-side guarantees that help to resolve inconsistency in data stores and the client-side guarantees that help to avoid causing inconsistency in data stores are discussed in Section 2.8. In Section 2.9 we describe the idea of adaptive consistency and its types.

2.1 Event Ordering

Consistency models are a set of rules that governs the execution of processes in a data store [TS06]. Processes consist of a set of events that can alter the current state of a data store or read the value of the current state. An event can also stimulate the execution of a process. Each process executes autonomously and independently from the other processes. But in order to define the correctness of the execution of the processes we need to identify the order of the events execution. The ordering of events depends on identifying the happened-before relationship.

Happened-Before Relationship The happened-before relation states the order of the produced events and help to execute processes in a consistent order on all data replicas. The happened-before relation is usually denoted by the symbol $\rightarrow$.

If a and b are two concurrent events and if a happened before b, then the happened-before relationship between the two events will be denoted as $a \rightarrow b$.

The happened-before relation satisfies the following three properties [Lam78] such as:

— If a, b are two events received by a node, and if a is received before b, then the happened-before relationship is denoted as $a \rightarrow b$.

— If a is an event of sending a message and b is an event of receiving the same message sent by a, then the happened-before relationship is $a \rightarrow b$.

— If $a \not\rightarrow b$ and $b \not\rightarrow a$ and the relation between the two events a and b could not be defined, then the two events are considered to be concurrent. This is denoted by $a \parallel b$.

Partial Order Identifying the happened-before relationship between one or more events defines the partial order between events. The happened-before relation $a \rightarrow b$ denotes the partial order $a \leq b$.

If S is the set of all events occurring in a distributed system, then the partial order relation between the events hold the following two properties [Lam78].

i Irreflexivity: $a \not\rightarrow a$, $\forall a \in S$.

ii Transitivity: $a \leq b$ and $b \leq c$, implies $a \leq c$, where a, b, c belong to S

Total Order While the partial order defines the happened-before relationship between only a set of events occurred in the system, a total order defines the happened-before relationship between all the events in the system. In total order any two events occurring in the system are ordered in some way and there are no two events that are concurrent ($a \not\parallel b$).

2.2 Time in distributed systems

In distributed systems, happened-before relations are identified via the notion of a clock time. Time in distributed systems is represented by an integer counter value given by a clock that respects two properties as described by Lamport in [Lam78]

P1: All clocks increment the counter value approximately at the same rate. The rate at which a clock runs at time t is $\frac{dC_i(t)}{dt} \approx 1$, where $|\frac{dC_i(t)}{dt}| - 1 < k \ll 1$.

P2: The difference between two clocks at some point in time is negligible, $C_i(t) \breve{} C_j(t) < \epsilon \ll 1$.

Based on the above two properties, time in distributed systems can be described in two types of clock:

— Physical Clock

— Logical Clock.

2.2.1 Physical Clock

Physical clocks are the electronic timers (clocks) that each computing node (computer) carries along. These clocks are based on the crystal oscillations frequency and are subject to clock drifts due to variation in oscillation frequency and temperature [CDK01]. Although the clock drifts are initially very small, eventually they add up to a larger number. Hence, periodic clock synchronization is very important in distributed systems that rely on physical clocks timing. However, clock synchronization is a continuous process as

the clocks tend to drift very often. Network Time Service Protocol (NTP) [1] is one of the popular clock synchronization protocol that compensates clock drifts over the Internet.

2.2.2 Logical Clock

Although it is possible to synchronize clocks across different machines to a single value, the value is still an estimation and it is hard to reach a more precise value. Moreover, most of the applications require to derive relative (causal) order of execution of the events rather than deriving order based on the time at which the events are executed. For this reason, the notion of clock called logical clock was introduced by Lamport in [Lam78]. Logical clocks capture the relative order of execution of the events more precisely than the synchronized physical clocks. Logical clocks are nothing but a distributed algorithm that captures the happened-before relationship between the communications between the various processes.

Logical clocks are based on two operations:

P1: Method to locally update the logical clock during the reception of new events.

P2: Method to synchronize two logical clocks during the reception of synchronization/ propagation/ gossip message from another node.

2.2.3 Lamport Clock

The initial model of logical clock was named after Lamport as *Lamport Clock*. In Lamport clock, each process stores a logical clock (say C_i), which is incremented as follows:

i Each time a process P_i receives an event, it increments its logical clock by one $C_i = C_i + 1$.

ii When a process sends a message m_k, it increments the logical clock by one ($C_i = C_i + 1$) and sends the C_i along with the message.

iii On reception of a message m_l from another process with a logical clock C_j, the process sets its logical clock to the maximum of its clock and the clock received along with the message $C_i = \max(C_i, C_j)$ and then increments the clock by one $C_i = C_i + 1$.

1. `http://www.ntp.org`

2.2.4 Vector Clock

One of the limitations of the Lamport logical clock is that it does not show the precise causal ordering of the events. For example, if $a \rightarrow b$, then $C_a \leq C_b$, but the reverse $C_a \leq C_b$ does not necessarily signifies $a \rightarrow b$. Hence in order to capture the causal ordering of the events more precisely, an enhancement of Lamport logical clock called Vector clock was introduced in [Fid88].

Vector clock consists of an array of N integers (say., $C_i[N]$), where N corresponds to the number of processes in the system. The clock $C_i[N]$ is updated as follows.

1. Each time a new event is received, the processes initialize or increment the corresponding integer in the logical clock by one. For example, process P_x does $C_i[x] = C_i[x]+1$ and process P_y does $C_i[y] = C_i[y] + 1$ and so on.
2. When a process P_x sends a message m_k, it increments its corresponding integer in the logical clock by one ($C_i[x] = C_i[x] + 1$) and sends the clock along with the message.
3. On reception of a new message m_l from another process with a logical clock C_j, the process compares each integer in its own clock (say C_i), and the clock received along with the message C_j and sets its clock C_i to the maximum of each integer between the two clocks. For all values of N, $C_i[n] = \max(C_i[n], C_j[n])$, where n ranges from $1 \,..\, N$, where N is the number of processes in the system.

2.2.5 Version Vector

Version Vectors are similar to the Vector Clocks with a small modification to the update rules. Both vector clocks and version vectors are widely used in replicated database systems. Like vector clocks, version vectors consist of an array of N integers, where N corresponds to the number of actors. An actor can be a replica node or a client who issues an update [PBA+10]. Unlike vector clock that corresponds the integer number in the clock to update events, version vector corresponds the integer number in the vector to the state of each replica.

Update rules of Version Vectors are as follows:

1. Each time an actor experiences an update, the corresponding integer in the version

vector is incremented by one. For example, Actor A_x does $V_i[x] = V_i[x] + 1$ Actor A_y does $V_i[y] = V_i[y] + 1$ and so on.

2. Each time a message is sent, the version vector V_i is sent along with the message.
3. While receiving a new message m_l from another actor with a version vector V_j, a version vector V_j will be updated to the maximum of each integer between the two vectors V_i and V_j. For all values of N, $V_i[n] = \max(V_i[n], V_j[n])$, where n ranges from $1 \,..\, N$, where N is the number of actors involved in the system.

2.2.6 Update-Conflicts

As mentioned earlier, logical clocks are used to signify time in distributed database systems in order to identify the happened-before relation between the events more precisely. The happened-before relationship helps to identify the order of the system events and apply the events on all the replica nodes in the right order. Applying the system events on all the replica nodes in the right order helps to achieve mutual consistency among all the replica nodes. Lamport clock is meant to identify the total order of all the events occurred on the system. However, sometimes two events that are received by different processes can have identical number. In that case, it is not possible to identify the order between those two events and hence both events are considered to be concurrent.

In multi-writer, multi-reader systems, where updates are applied in different order on different replica nodes, Vector Clocks and Version Vectors are widely used to identify the replica nodes that hold the recent version of a data item. Each data item consists of three fields including name, value and a vector clock or version vector. As mentioned in the previous section, both vector clock and version vector consists of array of N integers. If one of the integers in a vector clock (or version vector) is higher, and all other integers are the same as the other vector clocks, then the clock is said to be higher in the order. The data item whose clock is higher in the order will subsume the data items that have the same name and whose clocks are lower in the order. But in case where one of the integers is higher and not the same in all other integers, then neither of the vector clock (or version vector) subsumes the other and this is called as an update conflict.

For example: While comparing two vector clocks (or version vector) say $A[1, 4, 4]$ and

$A[1,3,4]$, the clock $A[1,4,4]$ will subsume the data item that has $A[1,3,4]$. But while comparing two vector clocks (or version vector) say $A[1,4,4]$ and $A[1,3,5]$, then neither $A[1,4,4]$ subsumes $A[1,3,5]$ nor $A[1,3,5]$ subsumes $A[1,4,4]$. This situation is called Update Conflicts.

2.2.7 Data Reconciliation

Data Reconciliation is the process of resolving conflicting states of a data replica via certain pre-defined reconciliation rules or with the help of application or client's support. Data reconciliation process can be broadly classified into two types such as Syntactic Reconciliation and Semantic Reconciliation. Syntactic Reconciliation uses causality or serializability relation between the update operation for conflict detection and resolution. Serializablity and Causality between update operations are normally identified via physical and logical clocks. Semantic Reconciliation on the other hand uses domain-specific knowledge for conflict detection and/or resolution. The input about domain-specific knowledge is usually given via specifying reconciliation rules for certain conflict types. If the reconciliation rules for a specific conflict type is not defined in advance, the system needs to get the user or system assistance for conflict resolution. In most of these cases, the system returns all the conflicting values during read operations and wait for the application or user to resolve the conflicts by emitting a new vector clock (or version vector) that dominates all conflicting updates.

2.3 Consistency models

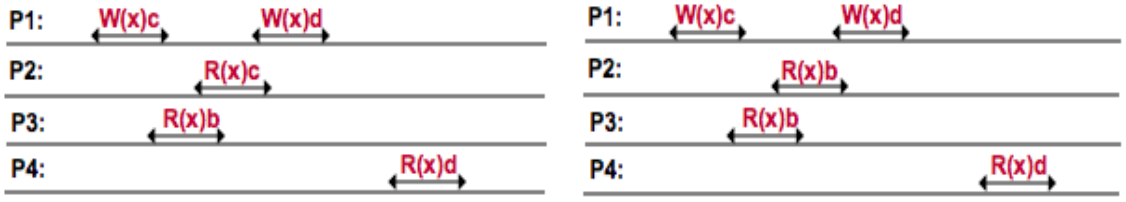
Following are some of the fundamental consistency models of distributed database systems.

2.3.1 Strict Consistency

Strict consistency aka Linearizability ensures that after a successful write operation, the value of the recent write will be visible to all other processes. In other words, the read and write operations happened on the system should follow the real-time order. The order

of the operations on the system expresses a total order corresponding to the wall-clock time.

The figure 2.1a shows the right ordering on the operations that respects Linearizability.
$W(x)c$ signifies write operation on data item x with a value c.
$R(x)c$ signifies read operation on data item x returns a value c.



(a) Order Compliant with Strict Consistency (b) Order Noncompliant with Strict Consistency

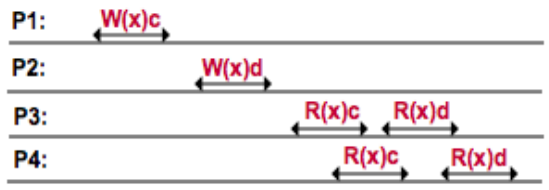
FIGURE 2.1 – Order that is compliant and noncompliant with strict consistency

From the figure 2.1a, we could observe that after the successful completion of a write operation $W(x)c$, Process $P2$ reads the value c. However, it is normal that the process $P3$ sees a previously written value b since the read operation on $P3$ happened before the completion of the write operation $W(x)c$. In the figure 2.1b, since the process $P2$ did not see the recent written value c after successful completion of $W(x)c$, the order does not accord with the linearizability.

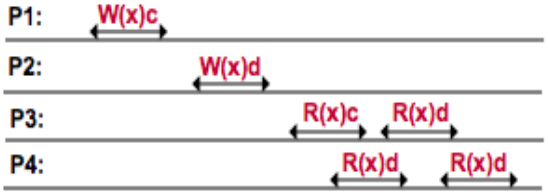
2.3.2 Sequential Consistency or Serializability

Sequential consistency was first stated by Lamport in [Lam79]. In sequential consistency, the operations Read and Write on the system follow a single global order across all processes, which does not necessarily respect the wall-clock time. That means, the client operations on the system follow the order specified by the application program. In Sequential consistency, the order of the operations on the system expresses a single global total-order as specified by the application. Hence, systems under sequential consistency can experience stale reads with respect to the wall-clock time, but not according to the logical clock.

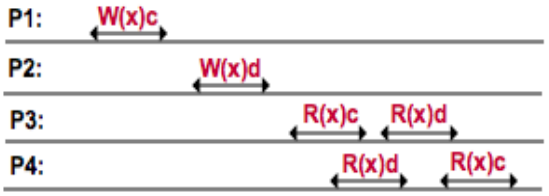
Figures 2.2a and 2.2b show an example of order of the system operations that accords to the Sequential Consistency model. From the figures, we can observe that the system accords



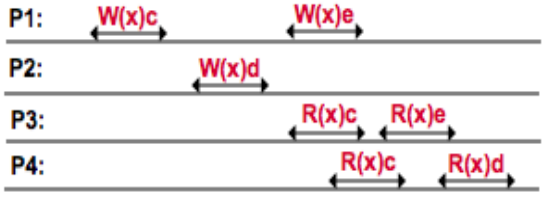
(a) Order Compliant with Sequential Consistency



(b) Order Compliant with Sequential Consistency



(c) Order Noncompliant with Sequential Consistency



(d) Order Noncompliant with Sequential Consistency

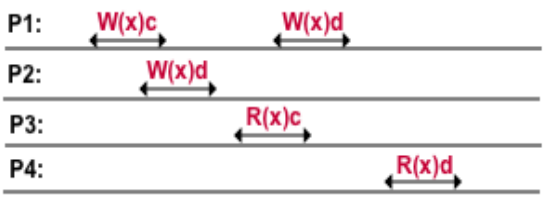
FIGURE 2.2 – Order that is compliant and noncompliant with Sequential Consistency

to a single global order of the operations following the order: $W(x)c, W(x)d, R(x)c, R(x)d$. Whereas the order of the system operations shown in the figure 2.2c and 2.2d do not accord to the Sequential Consistency behavior as the system does not follow a single global order. Figure 2.2c shows two conflicting orders such as $W(x)c, W(x)d, R(x)c, R(x)d$ and $W(x)c, W(x)d, R(x)d, R(x)c$. Figure 2.2d again shows two conflicting orders such as $W(x)c, W(x)e, R(x)c, R(x)e, W(x)d$ and $W(x)c, W(x)d, R(x)c, R(x)d, W(x)e$.

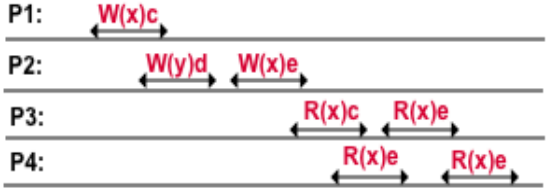
2.3.3 Snapshot Isolation

Both Snapshot Isolation and Sequential Consistency can be used for avoiding conflicting writes and achieve total order reads. Unlike Sequential Consistency that acquires locks for preventing concurrent writes, Snapshot Isolation avoids using locks for maximum throughput and prevents the concurrent writes during commit time by failing and rollbacking the conflicting write operations. In Snapshot Isolation, the system takes a snapshot of the current database state before the beginning of a transaction (set of read and write operations) and applies all the operations on the particular snapshot. By the end of the transaction, if one or more values that are affected by the transaction are also affected (and committed) by some other transaction or by an external update, the transaction has to fail and all the values that are affected during the transaction have to rollback to their

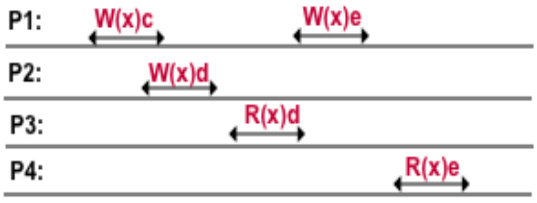
original state. Otherwise, the changes that are made during the transaction will be committed successfully. Snapshot Isolation is usually achieved via MultiVersion Concurrency Control (MVCC) protocol [CM86].



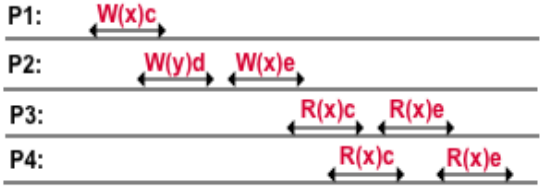
(a) Order Compliant with Snapshot Isolation



(b) Order Compliant with Snapshot Isolation



(c) Order Noncompliant with Snapshot Isolation



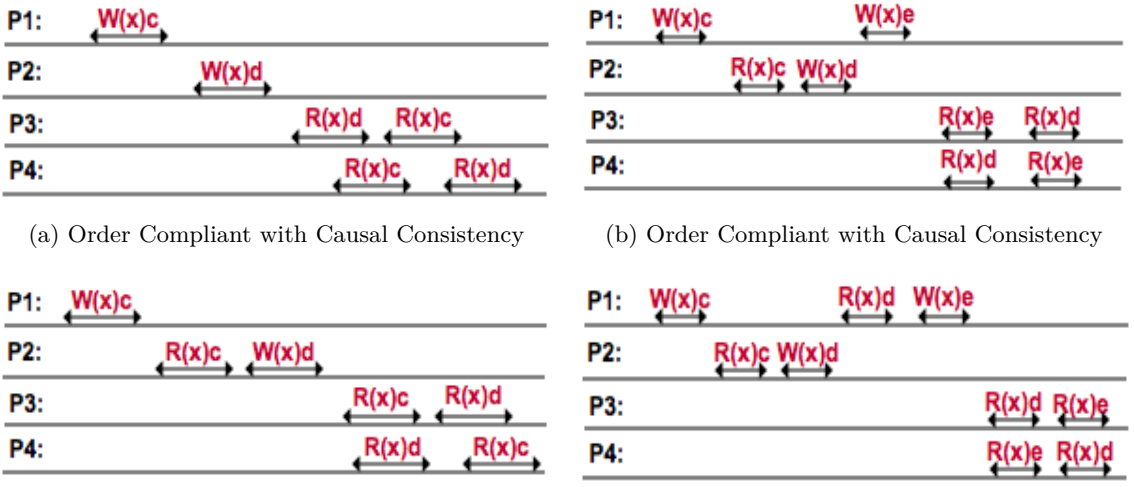
(d) Order Noncompliant with Snapshot Isolation

FIGURE 2.3 – Order that is compliant and noncompliant with Snapshot Isolation

From the figure 2.3a we could see that the write operation $W(x)d$ is started before the operation $W(x)c$ is committed to disk. Hence, the operation $W(x)d$ will start applying its changes to a copy of the database independent of the operation $W(x)c$. But, since the operation $W(x)c$ is committed to disk before $W(x)d$, at the end of the operation, when $W(x)d$ tries to commit its changes to the disk, the system will fail the operation. Therefore, any consequent read operation in the system will only see the effect of the operation $W(x)c$ and not see the effect of $W(x)d$. Hence the order shown in figure 2.3a is compliant with the Snapshot Isolation, whereas, the order shown in figure 2.3c is not compliant with Snapshot Isolation. In case of Sequential Consistency, the system avoids concurrent execution of the two write operations using locks. In figure 2.3d, since the operation $R(x)c$ on Process $P3$ is started before the operation $W(x)e$ on Process P2 commits to disk, it is correct to see the $R(x)c$ followed by $R(x)e$. But in Process $P4$, as the $R(x)c$ is started after the operation $W(x)e$ on $P2$ commits to disk, seeing $R(x)c$ is not compliant with Snapshot Isolation. The counter example shown in figure 2.3b shows the order that is compliant with Snapshot Isolation.

2.3.4 Causal Consistency

Causal consistency is a relaxed form of sequential consistency and snapshot isolation. Instead of respecting a single global order for all the operations on the system, in causal consistency, only the operations that are causally related follow a unique order. The operations that are not causally related are relaxed to be executed in a different order. The causality relations between the operations are captured via logical clocks such as Vector Clock and Version Vector.



(a) Order Compliant with Causal Consistency

(b) Order Compliant with Causal Consistency

(c) Order Noncompliant with Causal Consistency

(d) Order Noncompliant with Causal Consistency

FIGURE 2.4 – Order that is compliant and noncompliant with Causal Consistency

The figure 2.4a shows an example of a system order that is compliant with causal consistency but not compliant with sequential consistency. As the operations $W(x)c$ and $W(x)d$ are not causally related, the system under causal consistency is allowed to see the operations in different order. However, the order of the operations shown in the figure 2.4c is not causal compliant. As the operation $W(x)d$ follows $R(x)c$, the operations $W(x)c$ and $W(x)d$ impose a happened-before relationship $W(x)c \rightarrow W(x)d$. Hence, the system should not observe $R(x)c$ after observing $R(x)d$ and is not causally compliant.

In the figure 2.4b, the operations $W(x)d$ and $W(x)e$ are not causally related and so the system can observe the operations in a different order. However, the order of the operations in figure 2.4d shows a causal order between the operations $W(x)e$, $W(x)d$ and $W(x)c$ expressing a happened-before relation $W(x)d \rightarrow W(x)e$, $W(x)c \rightarrow W(x)d$ and by

transitivity $W(x)c \rightarrow W(x)e$. Hence, either observing the operation $R(x)c$ after observing $R(x)d$ or observing $R(x)d$ after $R(x)e$ are not causally compliant.

2.3.5 Eventual Consistency

Eventual Consistency ensures that any update operation made on the system will be eventually visible to all the replica nodes. Eventually consistent data stores are mostly Multi-Writer and Multi-Reader systems, where read and write operations succeed if at least one of the replica nodes acknowledges a request without any coordination. During update operation, an update initially modifies the state of one of the replicas and issues a success message to the client. The update will then be propagated in the background asynchronously ; eventually the update will be applied on all the replicas. However, the protocol does not ensure any order of visibility of the update operations. Due to the absence of ordering guarantees, different replicas may apply updates in different order and may temporarily conflict with each other: Update Conflicts. Update conflicts are common in eventually consistent systems and have to be handled separately during conflict-resolution phase [Vog09]. During conflict-resolution phase, the states of the conflicting replicas will be converged to a consistent state. The system ensures an inconsistency window time, which ensures, after a successful write operation, if there is no more future update, all replicas will converge to a same state. In these systems, all nodes are connected to a network where network partitions are unpredictably possible. In absence of network partition, the size of the inconsistency window can be estimated by calculating the communication latency between the replicas. Eventual consistency models are not advised for systems that are more prone to update conflicts. However, the benefits of this type of consistency model are Low Latency, High Availability and Partition Tolerance. The limitations are weak consistency guarantee, which include stale reads and update conflicts.

2.3.6 Comparison of consistency models

Figure 2.5 shows a comparison among the list of fundamental consistency models discussed above. The comparison is made in terms of the order of reads and writes, consistency guarantees, their performance and ease of programming. Linearizability, the stron-

gest among all the consistency options ensures real time reads and writes corresponding to the wall clock time. Programming distributed systems under this consistency guarantee is much easier but comes at the cost of performance. Serializability that ensures a single global serial order of reads and writes with a relaxation from accordance with the wall clock time is weaker than linearizability. Due to this relaxation, the performance of serializability is better when compared to linearizability, but the ease of programming is lower than linearizability. Snapshot Isolation that allows parallel execution of write operations for the favor of performance follows the execution order of the operations. Due to the possibility of some write anomalies such as write skew [SKS06], snapshot isolation is weaker than serializability. Causal Consistency that orders only the operations that are causally related exhibits higher performance, however its consistency guarantees are lower and even programming under causal consistency is more difficult. Eventual consistency model that does not follow any order ensures higher performance than any other consistency models. However, programming a distributed system under eventual consistency is highly challenging.

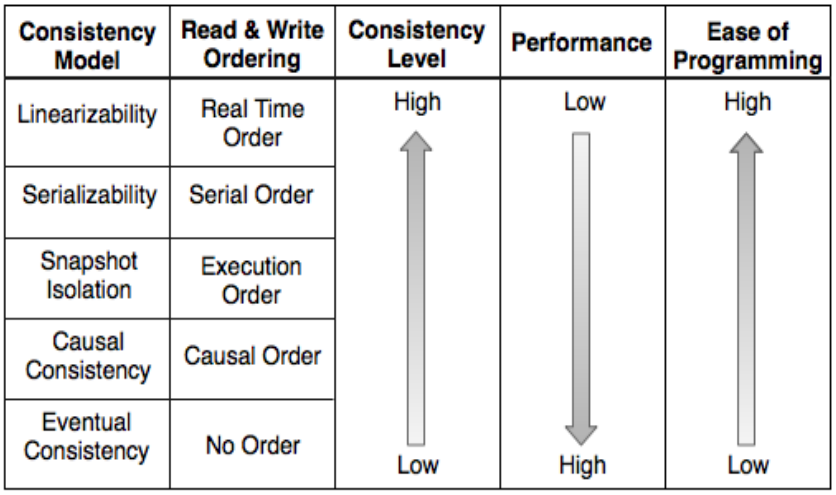


FIGURE 2.5 – Consistency Models Comparisons

2.4 Evolution of Modern Database Systems

The traditional database systems that are designed with ACID guarantees [Ram03] were poor in handling Big Data driven challenges. Thus, the revolution in database ma-

nagement led to the evolvement of modern database systems aka NoSql systems. In these databases, data items are distributed on various storage nodes connected to common network and often replicated on more than one nodes. These storage systems are usually non-relational. They rely on the BASE [Pri08] paradigm instead of ACID [Ram03] guarantees.

ACID Guarantees: ACID stands for Atomicity, Consistency, Isolation and Durability, the properties that are claimed to be desirable for traditional database systems. ACID offers high reliability for each operation happening on the system.

— Atomicity: Do all or Nothing. A transaction will succeed only if all parts of the transaction are successful. If a part of the transaction fails, the other parts should rollback to their original state.

— Consistency: The correctness of data accords to other application level semantics such as constraints, cascades, triggers etc.

— Isolation: Transactions have to be executed in an equivalent way as if they were independent.

— Durability: Once a transaction issues success message to a user, the changes should continue to persist irrespective of any physical node failures.

2.4.1 CAP influence

The CAP theorem [GL02b; FGC+97] has been highly influential in modern database systems, and is widely cited as a justification for the systems that rely on eventual consistency. CAP theorem states that a distributed system would demand three important properties namely Consistency, Availability and Partition Tolerance. But unfortunately, it is not possible to ensure all these three properties at the same time. A distributed system needs to sacrifice one of these three properties in order to ensure the other two properties.

The definition for Consistency, Availability and Partition Tolerance given by Gilbert and Lynch [GL02b] who formally proved CAP conjecture are as follows:

— Consistency: The clients should have a feeling of working on a single node regardless of the number of replicas.

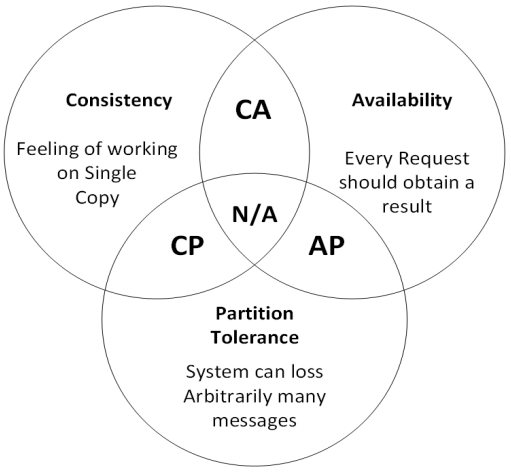


FIGURE 2.6 – CAP Theorem

— Availability: Every request sent by a client should obtain a successful response.

— Partition Tolerance: The system should continue delivering its services even if some part of the system loses many messages arbitrarily.

Abadi in [Aba12] argues availability will be affected only in case of network partitions. Network partition is not a feature of distributed systems, it is an unexpected event that may happen in a network. However, network partition is a rare scenario with modern technologies. Hence, sacrificing one among consistency and availability all the time, even if there is no partition in the network is a wrong approach. He also suggests CAP should be replaced by PACELC model [Aba12]. PACELC has two parts, one is PAC and another is ELC. PAC: if there is a Partition in the network, then the tradeoff should go between Availability and Consistency. ELC: Else, if there is no Partition in the network, then the tradeoff should be done between Latency and Consistency. PACELC can be written using the following equation 2.1.

$$Tradeoff \approx if(Partition)?AC : LC; \tag{2.1}$$

Table 2.1 shows the list of modern database Systems and how they conform to PACELC metric [Aba12; Mur13].

Storage Systems	In Partition		Normal Condition	
	Availability	Consistency	Latency	Consistency
Dynamo	⋆		⋆	
Cassandra	⋆		⋆	
Riak	⋆		⋆	
VoltDB/H-store		⋆		⋆
Megastore		⋆		⋆
MongoDB	⋆			⋆
PNUTS		⋆	⋆	

TABLE 2.1 – Distributed Storage Systems in the PACELC model

2.4.2 BASE Paradigm

BASE is an acronym for Basic Availability, Soft state, Eventual consistency. BASE is designed specially for attaining high scalability in distributed systems [Pri08]. It was proposed to promote rapid responses even when some replicas are not possible to be contacted.

— Basic Availability: All changes made by a user will be committed to the system without ensuring any guarantees about consistency and/or availability of the data even if some replicas are slow or crashed.

— Soft state: A data stored by a user can be lost unexpectedly in case of any individual node failure or system crash. Thus, the system does not ensure the durability of the stored data until certain window time, after which the data will be moved to 'hard state'.

— Eventual Consistency: After an inconsistency window time, all data replicas converge to a same state and reading from any of the data replica will see the effect of recent write operation.

As modern storage systems are inclined towards BASE paradigm, data consistency is not promised to the clients by default. As mentioned earlier, the fundamental justification given for this is the CAP theorem [GL02b]. Since it is very important for the distributed systems to be tolerant to partitions, they have no other choice than sacrificing Consistency in order to achieve high Availability [Aba12].

2.5 Tradeoffs of modern database systems

Modern database systems play different tradeoffs that favor targeted application needs. Two tradeoffs that are common in these systems are as follows.

2.5.1 Consistency-Latency Tradeoff

One of the important properties of distributed systems that is missed in the CAP conjecture is the request latency. The modern database systems relax consistency not only for the favor of High Availability and Partition Tolerance, but also for the favor of request latency. Abadi in [Aba12] illustrates different alternatives of data replication strategy and describes the consistency-latency tradeoff as inevitable in distributed database systems.

Propagating updates to all replicas at the same time and in the same order: Updating all the replicas at the same time is one of the traditional and straightforward approaches in propagating update operations. As each replica may process the updates in different order, sometimes the system may lead to update conflicts. In order to avoid update conflicts, the request will be passed through a preprocessing phase. In the preprocessing phase, the order in which all the replicas should apply the updates will be decided before applying the updates on the replicas. The advantage of this model is a simple and straightforward approach for ensuring strong consistency. The limitation of the model is the additional latency due to the preprocessing phase. This latency includes order agreement time and routing time especially when a replica is geographically far from the preprocessing node. This preprocessing phase is one of the typical constraints for the concurrent computing domain which is disclosed in the consistency model 'Sequential Consistency' [TS06].

Propagating updates from an agreed upon/ arbitrary node: The agreed-upon-node often refers to a master node or primary copy node. The main benefit of propagating updates from an agreed upon node is to prevent concurrent updates that lead to update conflicts. Instead of electing an agreed-upon-node and having a central path for update propagation, it is also possible to choose to propagate the update from any arbitrary

node. The advantages of this model over propagating updates from agreed-upon-node is no extra routing time and overhead for master node election. The limitation is that two updates for the same data can begin at different replicas and propagate the update to other replicas simultaneously (leading to update conflict). In this case, the system should detect and resolve the conflicts in addition to managing data consistency during read operations. In both cases (either propagating updates from an agreed-upon node or from an arbitrary node), the consistency guarantees of the read operations depend on how the update operations are propagated to other replica copies. The update propagation can take one of the three alternatives as follows [Aba12]:

Synchronous If the updates are applied synchronously, reading the data from any replica that is closer ensures strong consistency. However, while applying the updates, if one of the replicas is farther or slower due to some reasons, the latency of the update operation will be upturned by the slowest node responding to the request. Usually, in order to ensure strong consistency, the read access for the particular data will be blocked while the update is applying [SS05]. In this case, the read request has to wait or has to be dropped till the data is available for read operations. Thus, the availability of the system is compromised.

Asynchronous In the asynchronous propagation of updates, the node that receives the update first applies the update locally and returns success to the client. The update is then propagated to its replicas asynchronously in background. In this case the consistency-latency tradeoff depends on how the read requests are handled by the system.

— Case 1: If the read requests are served from a particular node as central point of entry, there won't be any sacrifice in data consistency. However, since the requests are served from a particular node, it has the risk of overloading the node and limits the read latency. A longer routing time in case of distant replicas also applies here.

— Case 2: If there is no central point of entry for the read requests, there is a risk of reading from a replica node where the recent update is not yet applied. In this case, the possible minimum read latency could be achieved, but in compromise to stale reads.

Hybrid Approach The combination of both Synchronous and Asynchronous propagation with intent to reach better Consistency-Latency-Availability tradeoff is also possible. This type of consistency method is known as Quorum consensus. The majority of replicas form a quorum. The size of the quorum can be estimated by the formula $N/2+1$, where N is the number of replicas for the data. The system will propagate the update synchronously to the Quorum and asynchronously to the rest.

2.5.2 Durability-Latency Tradeoff

The reason behind this tradeoff is disk seeks are expensive. During write operations if we write data to disk before returning a success message to the users, the writes will be durable but the write latency will be dominated by the disk seek time. In order to accelerate the write operations, modern database systems initially write data to an in-memory data structure and a issue success message to the user immediately. The write operations will be persisted to the disk eventually in the background. By this way the system moves the disk seek time away from the write path. Although this mechanism lacks durability guarantee, the system benefits from write latency and throughput. Thus the writes in modern database systems are initially soft state until the writes are written to the disk (hard state). This tradeoff is in accordance to S (Soft state) in the *BASE* paradigm. Some systems keep this option configurable by allowing the user to tune this guarantee on a per query or per table basis.

2.6 Replica Control Protocol

As consistency models are a set of rules between the processes and a data store, a replica-control protocol helps to preserve those rules in spite of various failure situations. The replica-control protocols depict the important properties of a system such as cost on read and write latency, system availability, load and scalability of the system. Replica-control protocols can be broadly classified into two types such as Primary-copy protocol and Voting-based protocol [Gif79].

2.6.1 Primary-copy Algorithm

In primary-copy (aka. master node) protocol, one of the replica copies, say r_i, among the set of replica nodes, say $r_1, r_2, ..r_n$, will be chosen as a primary-copy. The remaining replica copies other than the primary-copy are considered as the secondary copies. There can be a primary-copy for a set of data items or sometimes for the whole data set. The primary copies are usually elected by the replica nodes. In case of failure of the primary copy, the election will be re-conducted to choose one of the secondary copies as the new primary copy. In this model, first the update will be applied on the primary copy and then the update will be propagated in the same order to the secondary copies. Propagating the updates to the secondary copies can be either synchronous or asynchronous, sometimes a combination of both. Normally a replica-copy that is closer to the clients will be chosen as the primary-copy [LAEA95a]. In a multi-site system, the primary-copy would be chosen on the site, where a larger number of update operations originates [CRS+08].

If the primary copy node updates all the secondary copies synchronously, then the read operations can be forwarded to one of the closest replica copies and yield a consistent result. If the primary copy node chooses to update the secondary copies asynchronously due to performance reasons, then the read operation has to be forwarded to the primary copy to read the consistent result. However, as a tradeoff, a client can also read from any of the secondary copies that is closer to yield faster read response compromising stale reads. The main benefit of the primary copy protocol is that the model ensures the order of the updates and thus prevents update conflicts. The limitation of the model includes additional routing time when the agreed upon node is farther than one of the secondary copies and possible overhead in case of failure of the primary copy. The primary copy protocol is not well suited for write-dominant workload for scalability and performance reasons.

2.6.2 Voting-based replica control protocols

Quorum systems are well-studied in the literature [MRW97; MR97; AEA; Kum91; NW98; Vuk10]. In general, quorum systems consist of two or more subsets of nodes called

quorums: say $q_1, q_2, q_3 \cdots q_n$, where each quorum consists of one or more replica nodes such as $n_1, n_2, \cdots n_x$. Let us assume, a write operation is accomplished on all the members (nodes) of a quorum q_i, and a read operation is accomplished on all the members of the quorum q_j. If there exists at least one common member between the two quorums, q_i and q_j, then the read operation sees the value of the last write that is accomplished on quorum q_i. Having at least one common member in the two quorums where read and write operations are accomplished ensures the consistency of the system and this property is known as *Quorum Intersection Property*. Unlike quorum systems that determine the quorum members in advance, quorum-based voting systems determine quorum members based on the number of votes needed to succeed a query. For example, during write operation, a write will be forwarded to all the replica nodes and wait for the sufficient number of votes. The first n nodes that give sufficient vote become the quorum members. During read operation, the same technique could be followed, alternatively, since read operations are costlier than write operations due to disk seeks, the system can determine the quorum members in advance by choosing the first n closest available/reachable replica nodes. In distributed database systems, quorum-based voting systems are also known as NRW systems. In NRW, N represents the number of replica nodes of a data item, W represents the number of replica nodes contacted during write/update operation and R represents the number of replica nodes contacted during read operation. The system will exhibit strong consistency, if $W + R > N$ [Vog09; Vog08]. In addition, it is also essential to maintain $W > N/2$ in order to avoid update conflicts. Update conflicts are described more in the Section 2.7.8.1.

The process involves the following steps:

— During write operation, write a value for a data item D_x with a version number greater than the version number of the last write to all the members of a quorum q_i.

— During read operation, read the value of the data item D_x with the associated version number from all the members of a quorum q_j and choose a value that is higher in the version number.

— Having at least one common member between the two quorums q_i and q_j, so-called

quorum intersection property ensures consistency of the read operation.

2.7 Consistency guarantees of quorum based voting systems

In quorum-based voting systems, consistency guarantees are ensured based on the size (number of members) of the read and write quorums. The size could be specified explicitly to a certain number such as 1, 2, 3 or could be left implicit such as *all*, *quorum* (more than half).

Following are some of the popular consistency options of quorum-based voting systems.

— ONE signifies the size of the quorum is one. The read or write operations should be acknowledged by at least one of the replica nodes.

— ALL signifies the size of the quorum is N, where N signifies the number of replica nodes. The read or write operations should be acknowledged by all the replica nodes.

— QUORUM signifies the size of the quorum is $(N/2)+1$. The read or write operations should be acknowledged by the majority of the replica nodes, given by the formula $N/2+1$, where N is the number of replica nodes.

Based on the above consistency options, several consistency models have been proposed as follows and are described below.

2.7.1 Majority Quorum:

In majority quorum systems, the system uses consistency-option QUORUM for both read and write operations. Since both read and write operations are acknowledged by majority of the replica nodes (quorum), the system ensure the intersection property and guarantees consistency.

2.7.2 Weighted Voting:

Weighted Voting protocol [Gif79] is an earlier approach that described reasoning consistency via $W+R$ quorum overlapping [Vog12]. Unlike majority quorum systems that assign equal vote (usually one) for each replica nodes, weighted voting systems assign varied num-

ber of votes for the replica nodes. The system ensures consistency based on the formula $W + R > V$, where R denotes the number of votes obtained during read operation, W denotes the number of votes gathered during write operation and V denotes the total number of votes assigned for the particular data item. The protocol offers high availability and flexibility when compared to majority voting systems. By assigning a lower number of votes to the suspicious nodes that tends to fail often or get partitioned, the protocol will ensure intersection property with more number of node failures.

2.7.3 ROWA:

ROWA stands for Read-One, Write-All protocol [HHB02; BHG87]. As the name suggests, the read operations will be accomplished on only one of the replica nodes (consistency option ONE), whereas write operations have to be accomplished on all the replica nodes (consistency option ALL). The protocol is very efficient for read-heavy workloads. Since the read operations have to contact only one of the replica nodes, the clients can connect to one of the closest replica nodes benefiting the latency and availability of the system.

However, the protocol penalizes the write operations due to the necessity to contact all the replica nodes. In case of contacting all the replica nodes before issuing a success message to the client, the write latency will always be upturned by the slowest node responding to the request. In addition, if one of the nodes is down or unreachable, the availability of the system for the write operations will be affected.

2.7.4 ROWA-A:

In order to address the availability limitation of the ROWA protocol, ROWA-A is proposed [HHB02; Bur14]. ROWA-A stands for Read-One, Write-All Available. Unlike ROWA that rejects the write operations if one of the replica nodes fails to acknowledge, ROWA-A succeeds the operation as long as one of the replica nodes acknowledges the operation. The replica node that failed to acknowledge a particular write operation will be noted in the coordinator node as *hint*. The node that responds to a client request after querying the request from appropriate replica nodes is called *coordinator node*. When the replica node that missed a write operation joins back the cluster, the coordinator node

hand-offs the missed write operation using the noted hints. This process is popularly known as *Hinted Hand-off* [Wik13c]. One of the limitations of this protocol is that it does not ensure consistency in case of network partition. If a system is divided into two partitions and writes on one partition are not propagated to the other partition, then a client reading from one partition will not ensure data consistency.

2.7.5 Missing Writes Protocol:

In order to overcome the limitations of the ROWA-A protocol during network partition, the missing writes protocol combines both ROWA and majority quorum approaches. According to the missing writes protocol [ES83; HHB02], the system follows the ROWA strategy during normal situation and switches to majority quorum model if a communication failure is detected. The system follows the ROWA approach initially, and when a write operation is not acknowledged by one of the replica nodes, the system stores 'hint' about the missing writes and switches to the quorum consensus method. The information about the missing writes will be then communicated to all other nodes in the cluster whenever a communication between two nodes is established. So that during read operations, instead of reading from only one replica node, the system switches to read from majority of the replica nodes (majority quorum). However, for the data items for which the missing write information is not recorded, the system continues to use ROWA. The limitations of this approach include communicating the missing writes information to all the nodes in the cluster and even a temporary failure of single replica node affects the read performance.

2.7.6 Epoch Protocol:

One of the limitations of the ROWA and ROWA-A protocols is the write operations have to be accomplished on all the replica nodes, which in turn affects the latency and availability of the write operations. Although quorum consensus protocol minimizes this impact by contacting only a majority of the replica nodes, the protocol needs to contact the majority of replica nodes for both read and write operations. Hence, in order to address these limitations a new approach for replica control called epoch protocol is proposed in [RL93]. Epoch protocol is sometimes refereed as Highly Available ROWA (HA-ROWA) [Bas08].

Epoch protocol consists of a set of nodes called *epoch members*, which consists of a set of replica nodes that are trusted to be operational in the earlier period of time. During write operations, instead of communicating the operation to all/ majority of replica nodes, it is enough to communicate the operation only to the current epoch members (which are relatively a smaller set of replica nodes). And communicating the read operations to one of the members from the epoch ensures reading the value of the last write operation. However, the protocol has to run an epoch checking mechanism periodically in order to verify the failure of the epoch member. If one of the epoch members is failed, the protocol forms a new epoch that includes at least the majority of the previous epoch members.

2.7.7 Probabilistic Quorum

One of the main intentions of the quorum-based voting protocols is to ensure consistency with high availability despite of network partition (i.e., if one or more replica nodes are unable to contact). As a limitation, quorum-based voting protocol penalizes request latencies and the load on the system due to the number of nodes to be contacted during read and/or write operations. The idea of probabilistic quorum protocol [MRW97] is to relax the need for quorum intersection for some requests based on probabilistic evaluation. In other words, probabilistic quorum protocol satisfies intersection property only if the probability that a particular request leads to inconsistency is higher than some critical number. Several approaches exist in the literature to evaluate the probability that a particular request leads to inconsistency. In [BVF$^+$12] Bailis et al. evaluates probabilistic bounded staleness based on the time taken for a particular write operation to be propagated to all the replica nodes. If a read operation for the last written data item arrives before this probabilistic propagation time, then the read would be probably stale. Hence, based on the statistics about read/write pattern of the application or datastore, it is possible to decide whether a particular request needs to satisfy intersection property or not.

2.7.8 Partial Quorum:

Partial quorum systems that are based on eventual consistency use consistency-option ONE for both read and write operations in order to favor high availability and mini-

mum request latency. In this mode, users can experience data inconsistency. In order to ensure, strong consistency on demand, the user/application can use Consistency-option ALL either during read or write operation to satisfy the formula $W + R > N$ (Intersection property). For a *read-heavy workload*, choosing consistency-option ALL for writes and consistency-option ONE for reads would be beneficial. For a *write-heavy workload*, choosing consistency-option ALL for reads and consistency-option ONE for writes would be beneficial.

However, in distributed systems, networks are always unreliable, and hence, expecting to read/write from all the data replicas is a threat to data availability. If one or more replicas are down or unreachable, then we risk unavailability. So the safer way to ensure strong consistency on demand is to use Consistency-option QUORUM for both read as well as write operations. In this way, the intersection property will be satisfied without risking the system availability. One of the limitations of this approach is that both read and write operations must be communicated to a majority of replica nodes, which involves potential latency.

2.7.8.1 Consistency issues in partial quorum systems

Systems that choose read and write consistency-options based on the quorum intersection property do not run down into inconsistency issues. Systems that rely on eventual consistency and do not satisfy the intersection property, which are popularly called as partial quorum systems [BVF+12] mostly experience two types of data inconsistency. The two types of data inconsistency are as follows:

Stale Reads When a value of a data item D_x observed by an actor is different from the value observed by another concurrent actor for the same data item, then one of the two is subjected to stale read. The causes for stale reads could be either due to lost update or due to delay in the data replication as there is 'no now in distributed systems' as explained in [She15]. The eventual consistency guarantees of the system [Vog09] ensure that when there are no more future updates, stale reads will be automatically resolved by the system with the help of the version number associated with the put operations (cf. Section 2.6.2).

The version number identifies the state of a replica with respect to the particular data item, and it is normally captured via physical or logical clocks.

Update Conflicts As described in the Section 2.6.2, during write operation, a data item D_x is written with a version number that is higher than the previous version number of D_x. In order to do so, the replica node reads the local version number of the data item and then writes the new data value by increasing the version number by one. By maintaining the property $W > N/2$ (cf. ref 2.6.2) for each write, it is guaranteed that the version number of the new write will always be greater than the version number generated by a previous write operation.

For example, in case of 5 replica nodes ($N = 5$), let us assume, a write operation on a data item D_x is accomplished on 3 nodes say n_1, n_2, n_3 in order to ensure $W > N/2$. During the next write operation on D_x, even if the write operation is accomplished on the nodes that have a stale version of D_x, which are n_4, n_5, in order to satisfy $W > N/2$, the write operation has to be accomplished on at least one of the nodes among n_1, n_2, n_3 that have the recent version of D_x. The guarantee $W > N/2$ ensures an intersection between two successive write operations in order to avoid obtaining the same version number for two write operations. In some systems, a write operation will be first locally accomplished at one of the replica nodes and then the operation will be broadcasted to the other replica nodes before obtaining the sufficient number of votes. In that case, the possibilities of update-conflicts are higher. Update conflicts can be viewed as one of the side-effects of the lost-update or delay in data replication (stale reads).

However, systems like Cassandra [2] that uses high frequency timestamp as the version number do not have to satisfy the condition $W > N/2$. As timestamp is monotonically increasing, the timestamp of a new write operation will always be greater than the timestamp of the previous write operation.

2. `cassandra.apache.org`

2.8 Client-side versus Server-side guarantees

Since avoiding inconsistency during write and/or read operations affects the performance and throughput of the system, the process of resolving inconsistency can be moved to the background, away from the critical path. Inconsistency issues can be resolved or avoided based on some of the guarantees of server and/or clients.

2.8.1 Server-side guarantees:

In order to resolve data inconsistency in the system, server (nodes in the system) can adapt different mechanisms in the background as follows:

Syntactic Reconciliation: Syntactic Reconciliations are usually based on causality or serializability relations between update operations that are captured via physical clock or logical clocks (cf. Section 2.2). The clock information are used as the version identifiers of the data. When a data inconsistency is detected between replica nodes, the server tries to resolve the inconsistency by applying syntactic reconciliation. In syntactic reconciliation, the data whose version identifier (clock) higher in the order will replace the data that is lower in the order.

Read Repair: During read operations, the coordinator node can verify whether all the replica nodes contain the same version of the data item and initiate read repair in case of any mismatch. A read repair process involves the following steps.

— Retrieving the version identifier of the data item from all the replica nodes.

— Confirming whether the version identifiers retrieved from all the replica nodes are compliant.

— Updating the value of the replicas that have lower version identifiers with the value of the replica that is higher in the version identifier.

In partial quorum systems, where system usually read from only one replica node, the system can be configured to initiate read repair after a certain number of read operations on a particular data item [Gro15].

Active Anti-Entropy: The read repair process can resolve the inconsistency on data items that are frequently read by the clients. However, read repair can not resolve inconsistency on the data items so-called 'cold data' that are not read by the clients frequently. Anti-Entropy [Wik13a; RD15c] is an active background process for synchronizing the different states of the replica nodes. During this process, each node creates a Hash Tree aka. *Merkle Tree*, which is a tree of hashes representing the node's content and exchanges the hash tree between the neighboring replica nodes. If the hash trees of two replica nodes do not match, an inconsistency is detected between the replica nodes and will be resolved accordingly.

Conflict-free Replicated Data Types (CRDTs): As update-conflicts are common in systems that rely on Eventual Consistency, a stronger guarantee of eventual consistency, so-called Strong Eventual Consistency (SEC) that avoids update-conflicts, was introduced in [PMSL09]. Conflict-free Replicated Data Types are recent innovative data types that ensure Strong Eventual Consistency via deterministic merge functions that are based on the properties of Associativity, Commutativity and Idempotence. Designing a data type using these properties avoids update-conflicts in the system. CRDTs can be of two types based on the type of data replication used such as Convergent Replicated Data Type (CvRDT) and Commutative Replicated Data Type (CmRDT). In state-based replication, where replica nodes propagate their local states, designing a merge function that satisfies the properties of Associativity, Commutativity and Idempotence yields Convergent Replicated Data Type (CvRDT). In case of operation-based replication, where update operation is propagated instead of the replica state, designing operation execution as commutative with a guarantee of causal delivery yields Commutative Replicated Data Type (CmRDT).

2.8.2 Client-side guarantees

Apart from server-side guarantees that resolve inconsistency in the data store via various methods, client-side guarantees help to avoid inconsistency via different tactics as follows.

Read-your-writes: Read-your-writes model is one of the simplest and very useful consistency model for a large class of application scenarios. The model ensures the updated values are immediately visible to the user who issued the update and never see a version that is lower than the version he/she issued.

Session-consistency: Session-consistency set up a sticky connection to a replica node during a client-session. By doing so the system ensures read-your-writes and serializes the updates with respect to the client session.

Monotonic Reads: Monotonic-reads guarantees a user sees an increasing version of a data item. According to the approach, any successive read on a data item should return a newer version or a previously observed version. In order to do so, the client passes his/her last observed version of the data item along with each read operation, and the server ensures it does not return a data version that is lower than the one already observed by the client.

Monotonic Writes: Monotonic-writes guarantee an update issued by a process on a data item is successfully applied before applying an another update on the same data item by the same process.

Application-assisted Conflict Resolution The server usually resolves the data inconsistency between replica nodes by applying syntactic reconciliation using the version identifier attached with the data. But when the version identifiers of the data are not comparable due to update conflicts, the inconsistency has to be resolved by applying some domain-specific knowledge. In application-assisted conflict resolution, the application will be built with some domain-specific rules to handle these types of conflicts that couldn't be resolved by syntactic reconciliation. Hence, the server exposes the conflicting values to the application and the application resolves the conflicts by applying the business logic. This type of conflict resolution is referred as *Semantic Reconciliation*. Semantic Reconciliation is used as an alternative for syntactic reconciliation techniques where conflict resolutions could not be made based on the causality or serializability relations between the update

operations.

CALM Conjecture: CALM is an acronym for Consistency as Logical Monotonicity. CALM states that Strong Eventual Consistency (SEC) can be achieved without ordering constraints by designing the application via monotonic functions. Bloom [ACHM11a] is a programming language that is based on CALM conjecture. Bloom programming language helps programmer to identify the parts of the application code that could be built as monotonic logic and the parts that could not be built via monotonic logic. The parts of the application code where monotonic logic could not be applied need coordination service to enforce order.

2.9 Adaptive Consistency

Due to the distributed nature of modern database systems (NoSql Systems), ensuring data consistency always comes with a cost on request latency, availability and scalability of the system. In order to find a fine balance between data consistency, performance and throughput of the system, most of the modern database systems ensure eventual consistency by default and offer additional consistency options in order to ensure tight consistency on demand. These additional consistency options are relatively stronger than eventual consistency but add appropriate cost on the request latency and availability depending on the degree of consistency they ensure. Therefore, instead of relying on a single consistency protocol, mixing eventual consistency along with other supplementary consistency options makes the system more proficient. This phenomenon of using the appropriate consistency option depending on the criticality of the requests or data items is known as Adaptive Consistency. In this section, we first categorize the different types of data inconsistency in modern database systems and detail the types of adaptive consistency guarantees.

2.9.1 Categories of consistency models

Data consistency is a generic term that abstracts all types of possible errors that could be exposed to an actor. The actor can be anybody or anything that interacts with the

database via get and put methods. Remember, in most of the distributed storage systems, CRUD operations are achieved via just get and put methods, Create, Update, Delete as Put and Read as Get. The ACID properties of relational database systems helps to avoid all types of possible data inconsistency by ensuring 'Strong Consistency'. Since most of the distributed database systems are not ACID compliant for performance reasons, actors are prone to different types of data inconsistency. The types of inconsistency that an actor can observe from a modern database systems can be categorized as follows:

— Read-Read Inconsistency
— Write-Write Inconsistency
— Read-Write Inconsistency

Read-Read Inconsistency: In Read-Read inconsistency, the data value observed by an actor for a particular data item will be different from the data value observed by another actor for the same key. This inconsistency could be either due to a lost update of a replica node or the network-delay. Read-Read inconsistency is sometimes referred as *replication inconsistency* [SF12]. Read-Read inconsistencies are normally resolved automatically when the missing updates are applied on the replica nodes.

Write-Write Inconsistency: The Write-Write inconsistency or update inconsistency also known as update conflicts mostly arise as a side effect of Read-Read inconsistency. When an actor does an update for a data-key without reference to the most recent data-version, the system will be unable to identify causality between the data versions and considers both the versions as concurrent updates. In case of concurrent updates, by default, the system can not resolve the inconsistency automatically due to the conflicting replica states. Write-Write inconsistencies are normally resolved by exposing all the concurrent data-versions to the user/ application (actor) and get it resolved manually with the actor's assistance. The novel data types designed by Shapiro et al. called 'Conflict-free Replicated Data Types (CRDTs)' (cf. 2.8.1), help to avoid this type of inconsistency.

Read-Write Inconsistency: The third form of the possible inconsistency is Read-Write inconsistency, which is also known as Logical Inconsistency [SF12]. In traditional

database systems, read-write inconsistency is usually avoided via database transactions. A transaction consists of a set of update operations, which should be visible to the users only after all the updates are applied successfully, otherwise none of them should be visible to the users. A classical example for this consistency model is transferring money from one bank account to another. The process involves deducting money from one bank account and crediting it to another account. A user should see the balance of both the accounts either before the transaction or after the transaction was carried out. Reading the balance of the two accounts in an intermediate state is known as read-write inconsistency or dirty read.

2.9.2 Categories of adaptive consistency

From the literature, adaptive consistency techniques can be broadly classified into two types: User-defined and System-defined.

User-defined: In user-defined adaptability, the consistency level is tuned between each operations depending on the user/application needs. The operations can include any type of CRUD: Create, Read, Update, Delete operations. Based on the type or criticality of the operation, the user/application developer defines the desired consistency-level while querying the data store. User-defined adaptability is safer than system-defined adaptability as the consistency option is chosen by the users themselves, expressing stronger needs. However, it requires good knowledge of the application use case. Most of the modern storage systems such as Cassandra [LM10], Dynamo [DHJ+07a], Riak [Klo10], Voldemort [SKG+12] offer multiple consistency options and let the users/application to choose the needed consistency level based on the request severity. Existing consistency forms of this type in the literature are RedBlue Consistency [LPC+12] and SALT [XSK+14]. In RedBlue Consistency approach, operations are divided into two types namely RED and BLUE. The operations that are marked BLUE ensure weaker eventual consistency following asynchronous lazy replication strategy. Whereas, the operations that are marked RED ensure strong sequential consistency following synchronous active replication strategy. The operations that can commute to a same final state irrespective of the order

of operation execution will be marked as BLUE. And the operations where the order of execution could violate application invariance will be marked as RED.

SALT [XSK+14] approach combines the benefits of ease of programming with ACID guarantees and the performance benefits of BASE guarantees by introducing a new transaction model called *BASE Transactions*. The idea of BASE Transaction is to divide a ACID transaction into several mini subtransactions called *Alkaline Transaction*. The idea of dividing a transaction into multiple subtransactions is to leave space for more interleaving in a transaction. By making more interleaving in a transaction increases the execution performance by allowing higher concurrency. Also, it is possible to combine both ACID as well as BASE transactions in the same system selectively and preserve the atomicity and isolation of both transactions. The approach introduces a new level of isolation called *SALT Isolation* that ensures multiple granularities of isolation. In SALT isolation, an ACID transaction can observe a BASE transaction only when all the subtransactions (alkaline transactions) are completed. Whereas, the intermediate state of a BASE transaction can be observed by another BASE Transaction.

System-defined: One of the main challenges in the user-defined adaptive consistency is in categorizing the requests to a desired consistency level in advance. Deciding a consistency-level in advance becomes tricky for the application developers as the user and/or system behaviors may change dynamically over time. Adding intelligence to the system to predict the needed consistency-level based on the user/system behavior is the main scope of system-defined adaptive consistency. In system-defined adaptive consistency, the needed consistency-level to query data is identified dynamically during the run-time by finding a sweet point between consistency and inconsistency cost. Some of the factors that can be used to estimate the consistency during run-time are user's previous operations, data access patterns, value of the querying data, probability of update conflicts, network load, request latency, inconsistency cost and Service level agreement (SLA). This type of fine-tuning of the system consistency level can be done via system-defined adaptability. One of the famous consistency forms of this type that exists in the literature is Continuous Consistency [YV00]. Continuous Consistency approach defines the amount of staleness

that an application can tolerate from a particular data item. Based on the defined amount of staleness, the system adapts the consistency needs of the application. The staleness amount can be defined in three ways such as in terms of deviation in numerical values, deviation between the replicas and the deviation in the order of the update operations.

In [LLJ07] authors discuss about an infrastructure called *Infrastructure for Detection-based adaptive consistency (IDEA)* that helps to achieve Continuous Consistency. The work of Chihoub et al. described in [CIAP12] is also one such model that comes under Continuous Consistency model. The same author Chihoub et al. in [CPAB13] defines a consistency approach called Chameleon that helps to adapt the consistency needs of the application as a whole instead of adapting it according to individual data items. Chameleon uses machine learning techniques for modeling each application behavior and classifies different states of the application with respect to the application time-line. It applies the needed consistency option according to the application state. The work described in [WFZ+11] discusses an offline algorithm that quantifies the inconsistency observed by a client in a key-value store by merging the access traces from all system clients. The traces include information about operation start and end time along with the value stored or retrieved. The approach helps to decide an optimal consistency option for the system data. Sakr et al. in [SZWL11] describe uses of a middleware component called *CloudDB AutoAdmin* that sits in-between cloud data store and clients application and enable declarative specification of replica management and consistency adaptation. In [WYW+10] authors propose a system model that adapts the consistency needs of the application based on the read and update frequency of the clients.

A mix of both user-defined and system-defined approaches is also possible, as found in works on Consistency Rationing in [KHAK09]. Consistency Rationing divides the system data into three categories: Category A, B and C. The data that are marked with category A will always be handled synchronously to guarantee strong consistency. The data that are marked as category C, will be handled asynchronously and follows lazy replication. For the data that are marked as category B, the consistency need will be decided during run-time based on the additional environmental factors.

2.10 Modern Database System - Example

This section provides an example of a modern database system that can remain highly scalable and yet reliable in a large distributed environment where servers and network components are tending to fail often. Dynamo [DHJ+07a] distributed data store that is built by Amazon engineers for serving their largest e-commerce platform is one of the earlier systems of this kind. One of the main motivations for the development of Dynamo is to overcome the scalability and availability limitations of the existing Relational Database Systems (RDBMS). Relational database systems ensure correctness and high reliability in processing application queries with the guarantees of ACID properties (cf. Section 2.4). However, since ACID properties tend to affect availability of the system (cf. Section 2.4.1), and most of the application queries of amazon services can relax consistency over high availability, dynamo chose to move from ACID to BASE paradigm (cf. Section 2.4.2).

Dynamo targets applications that rely on simple read and write query model and can operate on weaker consistency guarantees. The main design motive of dynamo is to remain always writable, where any possible conflict (update conflict: cf. Section 2.2.6) could be handled during read time. The idea is to let customers always add or remove items to their shopping cart, and any inconsistency (or conflict) in the number of items added or removed is resolved later. In order to achieve that, dynamo follows an optimistic replication strategy with quorum-based voting technique for replica control (cf. Section 2.6.2). The number of votes that has to be collected in order to succeed a read or write operation is usually specified along with the application queries. In quorum-based voting technique, a write operation has to be accomplished by at least majority of the replica nodes in order to avoid update conflicts. Hence, a system can either follow ROWA (cf. Section 2.7.3) or majority quorum technique (cf. Section 2.7.1). But both options: Writing to All and Writing to Majority would affect the 'always writable' design goal of the system if sufficient number of replica nodes can not be contacted. In this case, in order to overcome temporary node or network failure, dynamo follows hinted handoff model as described in ROWA-A technique (cf. Section 2.7.4).

Apart from the tradeoff between consistency and availability, the system can also play

		Reference
Goal	Always Writable	[DHJ+07a]
Replica Control Protocol	Quorum-based voting	Section 2.6.2
Types of quorums	ROWA-A, Majority, Partial	Section 2.7
Tradeoffs	Consistency-Latency, Durability-Latency	Section 2.5
Data Versioning	Vector Clock	Section 2.2.4
Inconsistency Issues	Read-Read, Write-Write	Section 2.9.1
Server Side Guarantee	Merkle-Tree exchange	Section 2.8.1
Client side Guarantee	Application assisted resolution	Section 2.8.2
Adaptive Consistency	User defined	Section 2.9.2

Table 2.2 – Dynamo: Example of modern database systems

on two more additional tradeoffs: Consistency-Latency, Durability-Latency (cf. Section 2.5) in order to achieve high efficiency. Trading consistency over high availability and efficiency using partial quorum technique (cf. Section 2.7.7) leads to inconsistency issues: Stale Reads, Update Conflicts, which have to be handled separately. Dynamo uses both server-side as well as client-side support for handling data inconsistency and uses data versioning with vector clock (cf. Section 2.2.4). At server side, the system actively monitors the liveness of each node and makes sure nodes that are recovering from failure don't join back with stale data. In addition, the system employs active anti-entropy protocol using merkle-tree exchange (cf. Section 2.8.1) for detecting conflicting values of the replica nodes. As soon as a conflict is detected, the system tries to resolve it by applying Syntactic Reconciliation (cf. Section 2.8.1). The conflicts that could not be resolved at the server side via syntactic reconciliation will be resolved at the client side by applying the business logic via Semantic Reconciliation (cf. Section 2.8.2). Table 2.2 shows how the list of techniques that are discussed in the previous sections are used in dynamo.

2.11 Discussion

The stronger consistency options offered by the modern distributed database systems can take one of two possible variants. First one is reading or writing from/to all the replicas: Consistency-level ALL. Second is reading and writing to a majority of replicas: Consistency-level QUORUM. Some of the limitations of these models include extra cost

on request latency, high number of message exchanges and limits the system availability if one or few of the data replicas are cannot be contacted. In large distributed storage systems, it is more often that one or few replicas will be down at some point in time. Hence, in case of reading or writing to all the replicas, there is always a threat on system availability for a set of data items.

Most of these storage systems are optimized for write intensive workloads, which requires the system to acknowledge writes as fast as possible and reconcile possible conflicts during read time. In order to guarantee the reads with minimum latency, the non-overlapping quorum (eventual consistency) option is desirable for these systems. But, if a data item is written with the minimum write quorum (one node), the only mode to preserve consistency for this data during read time is reading from all the replicas. However, if one of the nodes is temporarily down or can not be contacted, the read will fail. The same risk applies when the system writes to all nodes and reads from one node. When reading and writing from/to a majority of nodes, the failure of one or few replica nodes is tolerable. In that case, availability guarantees will still be affected if the system is not able to communicate to a majority of replica nodes. Besides, the latency of both reads and writes will be affected.

Performance of these modern storage systems relies on caching most of the recent data in order to handle the read requests faster. When a request is forwarded to all or majority of replicas to retrieve a data item, the probability that all replicas have the right data in their cache would be low. Hence, the slowest replica node responding to the request will increase the request latency and the advantage of cache memory is lost.

Existing "strong" consistency options offered by these storage systems are strong enough to ensure data consistency when no partition occurs, but add some extra communication cost and a risk to data availability. To the best of our knowledge, there is no softer option that can ensure consistency with availability and latency guarantees similar to the default level of eventual consistency.

In this thesis we first explore the possibility of a new consistency protocol, which could act as an in-between consistency option between the default eventual consistency and the strong consistency options derived from the intersection property. The consistency protocol

that we designed called LibRe ensures consistency of the read operations while contacting a minimum number of replica nodes (ONE) during both read and write operations. LibRe protocol may coexist with different consistency options offered by the system and the system can apply the needed consistency option according to the user/application choice.

Although the system adapts its consistency guarantees by letting the user/application to choose the needed consistency option, deciding an appropriate consistency option for a query/data in advance during application development time remains challenging for the application developers. In order to overcome this challenge, the thesis proposes a system model, which helps to override the application-defined consistency options during run-time.

As LibRe protocol contributes to read-read inconsistencies, the thesis work also focused its contribution on write-write inconsistencies (update conflicts). In this contribution, the thesis discusses the challenges involved in reconciling the conflicting values of a data item and proposes a new data type called *Priority Register*. Priority Register is nothing but a Last-Writer-Wins Register data type that reconciles the conflicting values of a data item based on the application-defined replacement ordering instead of the physical or logical time (cf. 2.2). With the help of priority register, the slow and tedious client-side reconciliation process could be moved directly to the database side with significant benefits.

Chapter 3

LibRe: A New Consistency Protocol for Modern Distributed Database Systems

In distributed database systems, data is replicated to improve the performance and availability of the system. However, ensuring data consistency with higher availability and minimum request latency is notoriously challenging [Aba12; GL02a]. In order to efficiently handle these challenges, the Dynamo system [DHJ+07b] designed by Amazon uses a quorum-based voting technique that facilitates configurable tradeoffs between Consistency, Latency and Availability. This technique inspired subsequent distributed data storage systems such as Cassandra [LM10], Voldemort [Vol15] and Riak [Klo10]. The quorum-based voting technique ensures consistency based on the mathematical formula behind the intersection property of the quorum systems [NW98; Vuk10]. However, contacting a sufficient number of nodes in order to ensure the intersection property of the quorum comes with a communication cost and a risk to data availability. Although the systems enable the user or the application to configure the size of the read and write quorums according to the needs, if the intersection property cannot be satisfied, the system will reject the operations.

In this chapter, we discuss a new consistency protocol that we designed called LibRe, which tries to ensure consistency by contacting a minimum number of replica nodes thus reducing the communication cost and the risk to data availabilty. The protocol uses a registry that records the list of replica nodes containing the most recent version of the data items until all the replicas of the data item converge to a consistent state. Hence,

referring to the registry during read time helps to restrict forwarding the read requests to the replica nodes holding the most recent version of the needed data item. This chapter is organized as follows. In Section 3.1 we describe the general idea of LibRe protocol, following that, in Section 3.2 we describe the adaptation of LibRe protocol for modern distributed database systems including the system assumptions, architecture and working principles. Section 3.3 gives the formal description of the LibRe protocol. Section 3.4 discusses some of the works in the literature that shares few similar approaches as followed in the LibRe enhancement. And we provide a brief summary of the protocol at the end of the chapter.

3.1 LibRe

LibRe is an acronym for *Library for Replication*. The main goal of LibRe is to achieve stronger consistency while reading from only one replica copy irrespective of the number of replica nodes disconnected during reads and/or write operations. Systems that rely on eventual consistency protocols benefit from low request latency, high availability and scalability. One of the main limitations of the eventual consistency is its weaker consistency guarantees. In eventually consistent systems, at least one of the replica nodes contains the recent version of the needed data item. Thus, by avoiding to send the read requests to stale replica nodes (replica node that contains stale version of the needed data item), the consistency of the read operations can be preserved.

Hence, the design considerations of LibRe are:

— To ensure read requests are forwarded to a node that holds the recent version of the needed data item.

— To ensure system holds latency and availability guarantees similar to eventual consistency protocol.

3.1.1 LibRe Registry

One of the main challenges of the model is 'How to know a node is stale ?'. A stale node is the one where the recent update of a particular data item is not yet applied on it. We can not assume a node is stale if it contains one or few stale data items among a huge data set. The node is stale only for a particular data item, but will be consistent for the

remaining data items. Hence, the LibRe consistency model uses a registry to monitor each write and update operations and provides information about the replica nodes that holds the recent version of the needed data item. So that, the same node can be classified to be stale for particular data items and consistent for remaining data items. For example, if a particular node is down or unreachable from the network in any case for a certain period of time, the node will be inconsistent only for the operations that happened during this period. However, the node will be consistent for the remaining data items. Hence, the core idea of our approach is to identify the updates missed by a particular node. It enables the system to stop forwarding the requests to the stale node until it is consistent again. A node is considered to be stale if it contains stale data for the incoming request. This restriction can be freed once the lost updates are reapplied on the node. Hence, in order to achieve so, during update of a specific data item on a node, the node has to make an announcement to LibRe Registry. The announcement should be alike: the particular data with corresponding resource identifier is added/modified in this node. So that, with the Registry of nodes announcements, the Read requests will be forwarded to the node that contains the recent version of the needed data.

The next challenge of the approach is 'Where to keep the Registry ?'. The registry can be distributed among each node in the cluster or it can be on a single centralized node. LibRe protocol was tested in both centralized as well as in the distributed setup. We first describe the LibRe protocol in a *Centralized Registry Management*, followed by the description of LibRe in a *Distributed Registry Management* by sharding the registry across each node in the cluster.

3.1.2 Algorithm Description

Figure 3.1 shows the position of LibRe in the system architecture. LibRe is a centralized service that holds the three components *Registry*, *Availabilitiy Manager*, *Advertisement Manager*. The *Frontend* is a node in the system where the client connects to. In a Muti-Writer, Multi-Reader architecture, each node in the system can take the role of Frontend role. According to the LibRe protocol, write operations are handled in the same way as that of an optimistic replication protocol (write to one of the replica nodes and issue success

message to the client). Whereas, in case of read operations, the frontend node identifies a replica node that holds the recent version of the needed data item and forwards the read request to that node.

As shown in figure 3.1, LibRe protocol consists of three components namely *Registry*, *Availability Manager* and *Advertisement Manager*. The registry is a key-value store in which, for a given data-key, the set of replica nodes containing the recent version of the data item is stored. During write operations, the replica nodes notify the Advertisement Manager about the current update. Advertisement Manager in turn is responsible for updating the Registry recognizing the recentness of the data. The Availability Manager is contacted during read operations, and is responsible for forwarding the read request to an appropriate node that holds the recent version of the data.

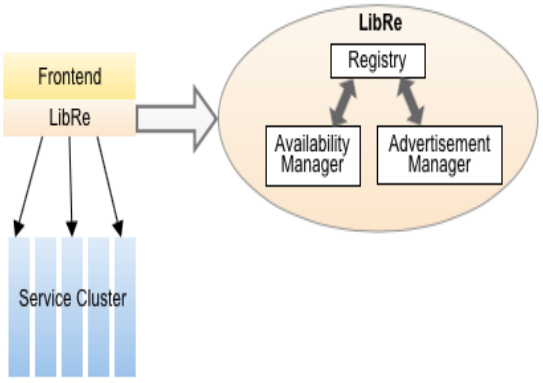
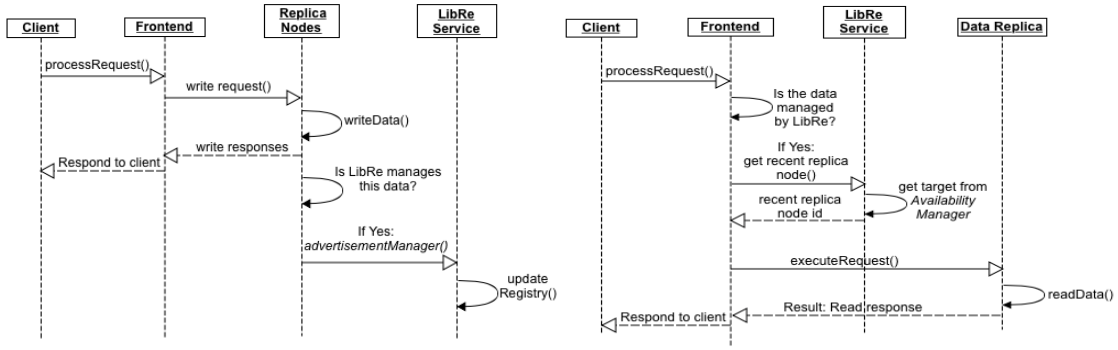


FIGURE 3.1 – LibRe General Architecture Diagram



(a) LibRe Write/Update Operation (b) LibRe Read Operation

FIGURE 3.2 – LibRe General Sequence Diagram

Figures 3.2a and 3.2b show the sequence diagram of a distributed data storage system that follows LibRe protocol for Write and Read operations respectively. From the figure 3.2a, when a client issues a Write/Update request, the frontend node forwards the request to one of the replica nodes that is responsible for the particular data item. When the write operation is successful on a replica node, the node checks whether the update has to be notified to the LibRe service. If yes, the replica node notifies the *Advertisement Manager* of the LibRe service about the update operation. Otherwise, no action will be taken. In the same way, when an update is propagated/gossiped to other replica nodes, each replica node after accomplishing the write notifies the *Advertisement Manager* of the LibRe service. These advertisements will be used to choose a right replica node for forwarding the read operations. During read operations, as shown in figure 3.2b, the frontend checks whether the information about the corresponding data key is managed by the LibRe service. If yes, the frontend node uses the LibRe service to find an appropriate replica node to query the needed data via the *Availability Manager* and forwards the request to that node. Finally, after querying the needed data, the replica node forwards the read response (result) to the client via the frontend node.

3.2 Adaptation of LibRe for Modern Distributed Database Systems

In this section, we will describe an enhanced version of the LibRe protocol with a distributed *Registry Management* while adapting the protocol according to the modern distributed database systems that follow Dynamo [DHJ+07b] bloodline. Instead of maintaining the LibRe components (Registry, Availability Manager, Advertisement Manager) as a separate service, this enhanced version of the protocol tightly integrates the components within each node in the cluster for better performance. These mechanisms are detailed in Section 3.2.4 along with the protocol description.

3.2.1 Targeted System

Key-Value data stores that store an opaque value for a given key have seen enough popularity in modern distributed database systems. Some of these systems ensure strong

consistency, whereas most of the systems such as Dynamo [DHJ+07b], Riak [Klo10] and Voldemort [Vol15] rely on tunable consistency. In order to tune the consistency level of the system on a per-query or per-table basis, these storage systems follow voting-based replica control protocol as described in Chapter 2. These systems mostly use a Distributed Hash Table (DHT) [ZWXY13] for identifying the replica nodes of a data item. LibRe protocol targets the Key-Value data stores that offer tunable consistency based on quorum based voting technique using DHT. In building LibRe, we assume that the underlying system provides failure detection and tolerance mechanisms, which are building blocks for the reliability of the LibRe Registry. Currently, we do not consider the use of LibRe for inter-cluster replication.

3.2.2 LibRe Registry

The core entity in the LibRe protocol is its 'Registry'. The registry is an in-memory key-value data structure that takes the data identifier as the Key and the list of replica nodes id (IP addresses) holding the most recent *version-id* of the data item as the Value. During write operation, each replica node tries to add its id in the list. If the number of ids in the list reaches the total number of replica nodes for the data item, then confirming the convergence of all replicas, the entry for the data item in the registry can be safely removed.

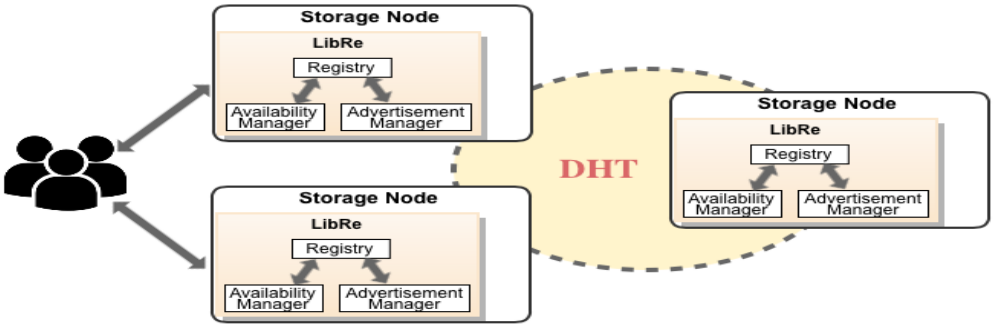
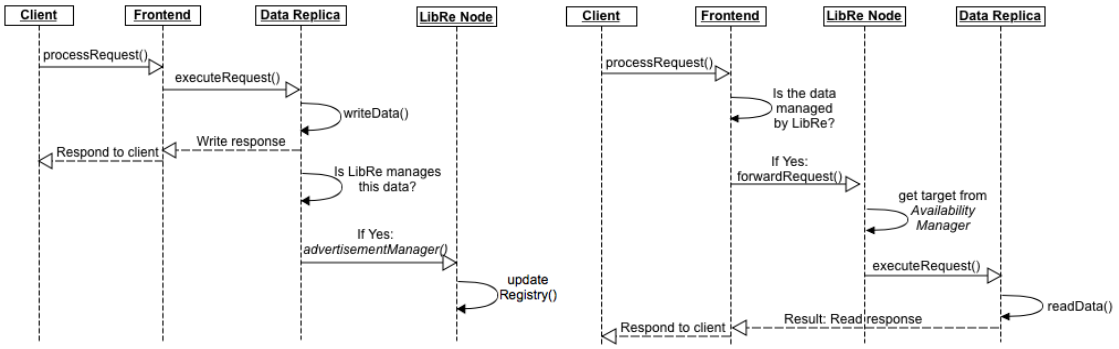


FIGURE 3.3 – Architecture Diagram of DHT-based LibRe

The registry is distributed over all the nodes in the cluster, but at any point in time, only one copy of the entry for a data item exists. An entry for a data item d_i will be stored on one of the available replica nodes that is responsible for storing the data item d_i.

Figure 3.3 shows the position of LibRe in the system architecture and the components of the LibRe protocol. Let R_i be a replica set for a data item d_i, such that $R_i = \{r_1, r_2, ..r_n\}$, where r_x is a node identifier, and n is the number of nodes in the replica set. So, one of the available replica nodes in R_i (say the first one: r_1) will hold the registry and the other two supporting components of LibRe: the Availability-Manager and the Advertisement-Manager, as shown in figure 3.3. The node that holds the registry and the supporting components is called the Registry Node or the LibRe Node for the particular data item. For any data item d_i, the id of the first replica node obtained via consistent hashing function [Kan14; KLL+97] is the registry node id. In other words, the replica node that has the lowest token id is considered as the registry node. The registry is distributed over each node in the cluster, so each node plays the role of replica node as well as registry node.



(a) LibRe Write/Update Operation (b) LibRe Read Operation

FIGURE 3.4 – Sequence Diagram of DHT-based LibRe

3.2.3 LibRe Messages

The LibRe protocol is based on two types of messages, namely: the Advertisement Message (figure 3.4a) and Availability Message (figure 3.4b), corresponding respectively to the Advertisement-Manager and Availability-Manager of the LibRe components shown in figure 3.3. The figure 3.4 and the corresponding algorithms 3 and 4 discussed at the section 3.2.4 show the improvements of the LibRe protocol over the original algorithm discussed in Section 3.1.2.

Advertisement Message: From figure 3.3, a client can connect to any node in the system. Some storage systems call this node the Coordinator node [LM10]. In the usual system behavior, a write request will be forwarded to all the replica nodes that are available. If the coordinator node receives back the required number of acknowledgements (or votes) from the replica nodes for the write, the coordinator issues a success response to the client. If the sufficient number of votes is not received within a timeout period, the coordinator issues a failure response to the client. LibRe protocol follows this default system behavior, but in addition, after a successful write operation, the replica node sends an *advertisement message* to the registry node asynchronously. The replica node sends advertisement messages only for the data items that are configured to use LibRe. The advertisement message consists of the data key, version-id and originating node id.

Availability Message: When the coordinator node receives a read request that is configured to use LibRe protocol, the coordinator sends an *availability message* to the Registry Node of this particular data item. The availability message contains the original *read message* received from the client and the data key of the needed data item. When the registry node receives an *availability message*, it finds a replica node from the registry and forwards the original *read message* to that replica node. The replica node sends the read response directly to the coordinator node and the coordinator forwards it to the client. If an entry for a data key is not found in the registry, then the *read message* will be forwarded to one of the available replica nodes.

3.2.4 LibRe Write Operation

Algorithm 3 describes the role of LibRe's *Advertisement Manager* during update operation. The update for a data item can be issued when its replicas are in converged state or in diverged state.

When a replica node sends an advertisement message regarding an update, the *Availability Manager* of the LibRe node takes on the following actions. First the protocol checks whether the *data-key* already exists in the *Registry*: line 3. If the *data-key* exists in the registry (replicas are in diverged state), line 3: the *version-id* logged in the registry for

Algorithm 3 Update Operation

$r_k = \{n_i, n_j, ...\}$: set of replica nodes holding recent version of data item k.
$e_k =< v_k, r_k >$: record where v is a version-id and r is a replica set.
$R : k \rightarrow e_k$: Map of data item keys k to corresponding entry record.
N: Number of replicas

```
1:  function ADVERTISEMENTMANAGER(k, v, n)
2:      entry ← R.k
3:      if entry ≠ ∅ and entry.v = v then
4:          entry.r ← entry.r ∪ {n}
5:          if |entry.r| = N then
6:              R ← R \ {k}
7:          end if
8:      else if entry = ∅ or v > entry.v then
9:          entry.v ← v
10:         entry.r ← {n}
11:     end if
12:     R.k ← entry
13: end function
```

the corresponding data-key is compared to the one sent with the update message. If the *version-id* logged in the registry matches with the *version-id* of the operation (replica convergence), then line 4: the node id (IP-address) will be appended to the existing replicas list. Line 5-6: If the number of replicas in the list is the same as the total number of replicas for data item k, then replicas are in converged state, and the entry is deleted. If the entry does not exist in the registry or the *version-id* of the operation is greater than the existing *version-id* in the registry (line 8): the entry is created or reinitialized with the operation's *version-id* and the sender node id (lines 9 and 10). Finally the entry is recorded in the registry (line 12). This setup helps to achieve Last Writer Wins policy [MSZ11; SS05].

3.2.5 LibRe Read Operation

Algorithm 4 describes the LibRe policy during the read operation. According to the algorithm, since the Registry keeps information about the replica nodes holding the recent version of data item k, the nodes information will be retrieved from the registry (line 2). Line 3: If an entry for the data-key exists in the registry, line 4: one of the replica nodes from the entry will be chosen as the target node. The method *first()* (line 4 and 6) returns the closest replica node sorted via proximity. Line 5: if the Registry does not contain an entry for the needed data-key, then, line 6: one of the replica nodes that are responsible for storing the data item will be retrieved locally via DHT lookup. Finally, the read message will be forwarded to the chosen target node: line 8.

Algorithm 4 Read Operation

$r_k = \{n_i, n_j, ...\}$: set of replica nodes holding recent version of data item k.
$e_k =< v_k, r_k >$: record where v is a version-id and r is a replica set.
$R : k \rightarrow e_k$: Map of data item keys k to corresponding entry record.
$D_k =< n_i, n_j, ... >$: replica nodes for data item k that are obtained via default method.

```
1: function getTargetNode(k, M_read)
2:     replicaNodes ← R.k.r
3:     if replicaNodes ≠ ∅ then
4:         n ← replicaNodes.first()
5:     else
6:         n ← D_k.first()
7:     end if
8:     forward(M_read, n)
9: end function
```

3.2.6 LibRe Reliability

As mentioned in Section 3.2.1, the reliability and fault tolerance of the LibRe protocol relies on the guarantees of the target system. The systems that use DHT for quorum based voting actively take care of the ring membership and failure detection [DHJ+07b]. The underlying DHT helps to find the first available replica node. In the event of node joining and/or leaving the cluster, the Consistent Hashing technique supports minimal redistribution of the nodes keys. In such case, there will be a change in the first available replica node (registry node) for a few data items and the registry information for these data-keys would not be available. In this case, the registry information will be rebuilt on the new registry node by sending successive *advertisement messages* to this node. This may lead to small and time limited inconsistencies in the system. Therefore, LibRe sacrifices Consistency in favor of Availability: cf. algorithm 4. If a registry node that has been unavailable joins back the cluster, the stale registry information has to be flushed during a handshaking phase. Besides, periodic local garbage collection is needed to keep the registry information clean between replica nodes.

3.2.7 LibRe Cost

The tradeoff provided by the LibRe protocol comes at the expense of some additionnal cost on message transfers and memory consumption.

3.2.7.1 Extra Message Transfers:

In LibRe, a lookup in the registry is required during a read operation on contacting the *availability manager* of the registry node to read from a right replica node. However, this operation represents constant cost, as the number of messages sent for achieving the consistent read does not depend on the number of replicas involved, as mentioned in section 3.2.3. Besides, the latency spent during this lookup can be gained back via managing the cache memory efficiently. During write operations, notifying the *advertisement manager* about an update is asynchronous and does not affect the write latency. Although these messages are an additional effort when compared to the default eventual consistency option, it is better than the strong consistency options that communicate synchronously to a majority or all replica nodes during reads and/or write operations.

3.2.7.2 Registry in-memory data structure:

LibRe manages the registry information in-memory. This information is distributed among all the nodes in the cluster and is maintained only for the data items whose recent update is not effected on all replica nodes. Moreover, eventual consistency guarantees of the targeted system and the periodic local garbage collection of the LibRe protocol helps to reduce the amount of information to be kept in-memory.

3.3 LibRe Formalization

In this section, we formalize the LibRe protocol and provide a proof of its reliability. This work requires the introduction of notations that will be used in the remainder of this section.

3.3.1 Notations

A data item is a pair of key and value $\langle k, v \rangle$, where $\mathcal{K}$ is the set of keys. The value of a data item k depends on time, so we must add this parameter to our model. Time can be physical or logical (version number), the only property needed on the set of times $\mathcal{T}$ is total ordering. We therefore introduce a *value function*, indexed by keys and time, whose

range is a set of values $\mathcal{V}$:

$$\begin{array}{rcl} value : \mathcal{K} \times \mathcal{T} & \rightarrow & \mathcal{V} \\ (k,t) & \mapsto & v \end{array}$$

We make the assumption, throughout this Section, that *all written values are distinct*, this can be obtained by tagging values themselves with the time of the write request, so considering $\mathcal{T} \times \mathcal{V}$ instead of $\mathcal{V}$ as the set of values.

Thus, a data item *at time t* is the pair key/value $\langle k, value(k,t)\rangle$, denoted as k_t, which is the *latest version* of k at time t. We consider a *domain* $\mathcal{D} = \{k^1, ...k^m\}$ of items (represented by their keys) managed by LibRe. We also define the *latest modification time* with repect to time t, as:

Definition 1. *Given a key k and a time t, we let $latest(k,t) = \min\{t' \in \mathcal{T} \mid k_{t'} = k_t\}$.*

In particular, for any $t'' \in \mathcal{T}$, if $latest(k_t) \leq t'' \leq t$, we have $k_t = k_{t''}$. This includes $t' = latest(k_t)$ itself and this is the minimal such time, for which $k_{t'}$ is still the latest version of k (at time t). This is directly related to the last write request and represents an idealized model of storage. We will also denote $latest(k,t)$ the current data itself, $\langle k, value(k,t)\rangle$, which is the latest version of the data, that the system should provide when requested by the user. We can also compute a *version number*:

Definition 2. *Given a key k and a time t, we let $version(k,t)$ be*

```
function VERSION(k, t)
    if t = 0 then
        return 0
    else
        t' ← latest(k, t)
        return (1+ VERSION(k, t' − 1))
    end if
end function
```

As written values are distinct, we can define the write time of a value $v \in \mathcal{V}$:

Definition 3. *Given a key k and a value v, we let*

$$time(k,v) = \min\{t \in \mathcal{T} \mid value(k,t) = v\}$$

t is uniquely determined, and we have in particular $time(k,v) = latest(k, time(k.v))$. It can be undefined in case v was never associated to k. This also implies a new definition of the version number:

Definition 4. *Given a key k and a value v, we let $version(k, v) = version(k, time(k, v))$.*

A replica node, denoted r, contains a partial map from key to values, that depends on time. So its type is:

$$\mathcal{T} \to (\mathcal{K} \hookrightarrow \mathcal{V})$$

The state of the map at time t is $r(t)$, also denoted r_t, and dependency on time is usually left implicit. It is a partial mapping (notation $\mathcal{K} \hookrightarrow \mathcal{V}$) since it is undefined on keys it is not in charge of. A read operation for k on that map is then $r_t(k)$ and a put operation is denoted $r_t[k \leftarrow v]$. It has the effect that r_{t+1} is the same map as r_t, except on k.

Remind from Section 3.2.2, that the set of replica nodes in charge of an item k is given by applying the consistent hasing function $h(k) = \{r_1^k, ..., r_n^k\}$. We assume that this set is a *list with no repetition*, so the Registry Node for k is r_1^k, more compactly noted r_k, when no confusion arise.

The registry of a node r is called $\rho = reg(r)$. ρ bears the same indices as r, in particular, the registry responsible for k is $\rho_k = \rho_1^k = reg(r_1^k)$. Registries also vary over time, therefore are function of the following type:

$$\mathcal{T} \to (\mathcal{K} \hookrightarrow \wp(\mathcal{R}))$$

$\rho(t)$, as usual noted ρ_t or ρ when the dependency on time is implicit, is a *partial* function. It associates a subset of $\mathcal{R}$ (the nodes) to *some* keys, at most the ones the node r is in charge of. As discussed in Sections 3.2.4 and 3.2.5, even $\rho_k(k)$ may be undefined, or more generally $\rho(k')$ for any arbitrary k' the registry ρ is in charge of. The intended meaning is that, in such a case, the Registry Node *knows* that the replicas for k' are in a consistent state. We prove this claim later in the Section.

We also need to introduce the function R_l, that to each data item k associates the *actual* set of replica nodes that contain the latest version of k (at a given time t):

$$\begin{array}{l} R_l(k,t) \subseteq h(k), \\ r \in R_l(k,t) \text{ if and only if } r_t(k) = value(k,t) \end{array}$$

As we are assuming that all written values are distinct, it cannot be the case that $R_l(k, t)$ contains some stale node r, for which $r_t(k) = value(k, t')$ for $t' < latest(k, t)$, but still enjoying $value(k, t') = value(k, t)$.

By *definition* of $R_l(k,t)$, the following holds: $time(k, r_t(k)) < t$, for any $r \in h(k) \backslash R_l(k,t)$, i.e. $r \in h(k)$ but $r \notin R_l(k,t)$. Otherwise, r would belong to $R_l(k,t)$. Otherwise said, $d \in r \Rightarrow version(k, r_t(k)) < version(k,t)$ for those r not in $R_l(k,t)$.

The operations are of two kind, get (g) or put (p). They arrive at various time, as a discrete stream. Each operation contains: a time t; the data key k; and put operations also contains the new value v. We access those value with a dot: $p.t$, $p.k$ and $p.v$. All put operations p enjoy the following property:

$$value(p.k, p.t) = p.v$$

We also assume, for any key k and time t (note that $p.t$ is different from t):

$$\text{if } latest(k,t) \neq 0 \text{ then } \exists p, p.k = k \text{ and } p.t = latest(k,t) \text{ and } p.v = value(k,t)$$

Lastly, we assume *no interference* of a put operation p over the state of the system (registries, data on replica) associated with a key $k' \neq p.k$.

3.3.2 System States

The system can be in one of the following two states with the associated properties.

3.3.2.1 Stable State: S1

The system is said to be in stable state at t if two conditions are met. The first one is that there is *no change* in ring membership compared to $t-1$, and the second one is that the system enjoys (at t) for any k the two following properties:

$$\rho_k(k) \text{ defined } \Rightarrow \varnothing \neq \rho_k(k) \subseteq R_l(k) \tag{3.1}$$

$$\rho_k(k) \text{ undefined } \Rightarrow h(k) = R_l(k) \tag{3.2}$$

As those Properties 3.1 and 3.2 are mutually exclusive, we will ensure only one of them at a time for a given k. In case either property holds for k, we say that the registry for k is *up to date*, or simply that the registries are up to date, if this holds for all $k \in \mathcal{K}$. Note that in Property 3.1, ρ_k can be *strictly smaller* than $R_l(k)$, in case there was a communication

problem (cf. Lemma 1). Moreover, in state S1, we know that the last version of k is present on some replica node, and accessible to the other nodes. As a consequence:

$$R_l(k) \neq \emptyset \tag{3.3}$$

3.3.2.2 Unstable States: S2, S3, S4 and S5

We denote the state, when a node r *leaves unexpectedly the ring* or becomes unaccessible, at time t, by S2. In this case, there is a switch of the registry for the following set of keys:

$$sup(\rho_{t-1}) = \{k \text{ such that } h_{t-1}(k) = \{r, r_2, \cdots, r_n\}\}$$

$sup(\rho)$ is the support of ρ, the set of all keys it can be defined on. It is larger than $dom(\rho)$, the *domain* of ρ, which is the set of keys on which ρ is currently defined. An undefined $\rho_{t-1}(k)$ for $k \in sup(\rho) \backslash dom(\rho)$ just means that $R_l(k, t-1) = h(k)$ (replicas have converged). For such a key k, removing r has virtually no impact. Generally speaking, we get, for $k \in dom(\rho)$:

$$\rho_k(t)(k) \neq R_l(k, t)$$

In more details, fix some key k such that $h_{t-1}(k) = \{r, r_2, ..., r_n\}$. As r has left the mode, $h_t(k) = \{r_2, ..., r_n\}$, and $\rho_k(t) = \rho_2(t)$. So $\rho_k(t)(k) = \rho_2(t)(k)$, which is either undefined, or contain old stale information. If we had $R_l(k, t) = h_{t-1}(k)$, which corresponds to $k \notin dom(\rho)$, we are safe since $\rho_2(t)(k) = \rho(t-1)(k) = \varnothing$ or $\rho_2(t)(k) \subseteq h_t(k)$. We are also safe if $\rho_2(t)(k) \subseteq \rho(t-1)(k)$ or if $\rho_2(t)(k) \subseteq R_l(k, t)$.

But if we are not safe, then Algorithm 4 will pick any of the nodes of $\rho_2(t)(k)$ or $h(k)$, with a risk of staleness with probability that depends on $|R_l(k)|/|h(k)|$, see Section 3.3.4.

However, in state S2, for any k, in particular those in $dom(\rho)$ the following property holds, which is formally proved in Lemma 1.

$$\forall p, (p.t > t \wedge p.k = k) \Rightarrow \forall t' > p.t + \tau_{adv}, R_l(k, t) = \rho_2(t)(k) \tag{3.4}$$

This means: as soon as we have a put operation on k, at any time after at least one advertisement message has been received (hence the delay τ_{adv} added to the time $p.t$ the

request was made), the registry ρ becomes up to date for k (provided other errors do not occur in the frame $[t, p.t + \tau_{adv}]$).

We denote the state when a node joins the ring by S3. In this case, there is also a switch on the registry for certain keys k. The system *can* fall back in the state $\rho_k(t)(k) \neq R_l(k,t)$, especially if the new node joins the ring without synchronization. The system enjoys the same property 3.4.

The last unstable states, S4 and S5 are similar: the Registry Node r receives no advertisement message and the Coordinator Node c receives the required number of acknowledgements (S4), or r receives at least one advertisement and c receives not enough acknoledgements (S5). Those states are unlikely to happen, as it is rather an indicator that nodes are unacessible (state S2). All states S2 through S5 produce same effect on the system, that is to say:

$$S2 \approx S3 \approx S4 \approx S5$$

There are other situations, when a write operation fails. One last instance is when no replica is available, in which case the Coordinator will issue a failure, so this does not produce any consistency issue.

As for the likeliness of state S4, a put operation is sent to all the replica nodes in parallel (including r_k), and the local advertisement message is likely not to be lost. In the case where the put operation sent to r_k is lost, the registry ρ_k will be updated on reception of at least one of the advertisements sent by the replica nodes that have successfully voted to the Coordinator (assuming that if the Coordinator receives the acknowledgement, the Registry receives the advertisement). In the worst case, the put operation sent to r_k is lost and all the advertisement message sent to r_k by the replica nodes, that succeeded, are lost as well, and we fall in state S4.

As for state S5, such a situation might arise more frequently. This is less harmful, since the nodes pointed to by the Registry Node for k would contain newer data, than the one given by the last successful write.

3.3.3 Stable State Properties

We assume to be on a time range $[t_1, t_2]$, such that there is no change in the ring membership during the time frame $[t_1, t_2]$. As well, we assume that all the put operations received in $[t_1, t_2]$ are sucessfully advertised at least once to the Registry Node *and* that the Coordinator issues a success response to p. Otherwise, we fall in one of the unstable states, discussed in the next section.

Given a put operation p, we define τ_{adv}, abbreviated as τ, the delay with which the *first* advertisement message is received by the Registry Node. As assumed there is at least one such message.

We show that the users always read the most recent version of any data item k, except during the time when a put operation is in process on k. We begin with lemmas *that do not depend on the state*, and that we will reuse later. The next lemma does depend only on the success of p (and on no change on ring membership):

Lemma 1. *Consider a put operation p. $\rho_{p.k}$ is up to date for $p.k$ at time $t + \tau_{adv}$, provided that we do not receive another put operation p' on k in $[p.t, p.t + \tau_{adv}]$.*

For simplicity, we assume no concurrent put operation p' in the interval between the reception of p and the first advertisement is successfully sent to $r_{p.k}$. In practice, LibRe takes care of overlapping put operations by applying the Last Writer Wins (LWW) policy with the help of the tie breaking mechanism provided by the underlying system. To take this into account, we need only to consider a set of overlapping put operations, and ensure up-to-dateness wrt the last one.

Proof. During the operation, the Coordinator sends the update in parallel to all the replica nodes in $h(p.k)$ and waits for sufficient number of votes (nodes acknowledgement for successful write) according to the chosen consistency option. A success message is issued to the writer as soon as the threshold is met (at time $t' > p.t$).

We assume that $h(k)$ contains more than one replica. By Algorithm 3, $\rho_{p.k}(p.k)$ is set to $\{r\}$, where r is the source of the first advertisement message, received at $p.t + \tau$. Therefore $\rho_{p.k}(p.k)$ is not empty, and by definition: $r \in \rho_{p.k}(p.k)$ implies data $\langle p.k, p.v \rangle$ has been

written.

Since there is no overlapping put operation on $p.k$, $p.v = value(p.k, p.t) = value(p.k, p.t+\tau)$, therefore $r \in R_l(p.k, p.t+\tau)$ and $\rho_{p.k}(t+\tau) \subseteq R_l(p.k, t+\tau)$. Property 3.1 holds for k.

Lastly, if $h(k)$ contains only one replica, then it must be r, the advertiser. We know, by the same reasons as above, that $r \in R_l(p.k, p.t+\tau)$, as a consequence $R_l(p.k, p.t+\tau) = h(k)$. $\rho_{p.k}(t+\tau) = \varnothing$ by the algorithm, which means, that Property 3.2 is respected.

This covers all the possible case that can happen during the execution of a successful put operation, including loss of put messages sent to the $r \in h(p.k)$ (including $r_{p.k}$ itself), of advertisement messages sent to $r_{p.k}$, and of acknowledgement messages sent to the Coordinator. The important point is that the Registry Node is up to date as soon as one advertisement message is received. □

The next lemma says that up-to-dateness is hereditary in time. It does not depend on the success of any put operation, nor on the state we are in, and only assume stability of ring membership:

Lemma 2. *Let k be a key, assume that ρ_k is up to date for k at t. Let $t' \geq t$. If no put operation on k is received between t and t', then ρ_k is up to date for k at t'.*

Proof. This proof is by induction on t':

— if $t' = t$, then ρ_k is up to date (for k, left implicit from now on) at t' by assumption.

— assume that ρ_k is up to date at t', let us show that it is up to date at time $t'+1$.

Several cases must be distinguished:

— $\rho_k(t')(k) = \varnothing$. Then all the nodes of $h(k)$ contain the last version of k, by Property 3.2. As no other put operation on k is received, $R_l(k, t'+1) = h(k)$.

— Most likely $\rho_k(t')(k)$ is undefined because all advertisement messages have already been received (Algorithm 3), so we just have $\rho_k(t'+1)(k) = \varnothing$.

— But we might be in a stable state for other reasons (coming from an unstable state), in which case some advertisement may be received at time $t'+1$. By the algorithm, $\rho_k(t'+1)(k) \neq \varnothing$ (it becomes defined). In this case, we still are

safe, since Property 3.1 is respected, as $\rho_k(t'+1)(k) \subseteq h(k) = R_l(k, t'+1)$.

— $\rho_k(t')(k) \neq \varnothing$. Then we distinguish the following cases:

— we receive no advertisement at $t'+1$, or a stale advertisement: $\rho_k(t'+1)(k) = \rho_k(t')(k)$ is still up to date.

— we receive a valid advertisement from a node r. ρ_k will be updated. As $\rho_k(t')$ was up to date, we have by Property 3.1, $\rho_k(t')(k) \subseteq R_l(k, t')$. We also know that, at $t'+1$, r has succesfully written the data, so $r \in R_l(k, t'+1)$. Lastly, $R_l(k, t') \subseteq R_l(k, t'+1)$. This implies $\rho_k(t')(k) \cup \{r\} \subseteq R_l(k, t') \cup \{r\} \subseteq R_l(k, t'+1)$.

We now distinguish two possibilities. First, if $\rho_k(t')(k) \cup \{r\} = h(k)$, we set $\rho_k(t'+1)(k) = \varnothing$. But this also implies, by the results just proved, that $h(k) \subseteq R_l(k, t'+1)$. Thus, $R_l(k, t'+1) = h(k)$ (it cannot be larger) and Property 3.2 is respected. Otherwise, $\rho_k(t'+1)(k) \subseteq R_l(k, t'+1)$ and Property 3.1 is satisfied.

□

This allows us to derive preservation of stability of a system on the interval $[t_1, t_2]$.

Lemma 3. *Assume that the system is in stable state S1. If we receive only successful put operations, then the registries are up to date at any time $t \in [t_1, t_2]$, if no put operation is ongoing at t. In other words, the system is in stable state S1, except when a put operation is ongoing.*

Proof. Fix some key $k \in \mathcal{K}$. We first prove by induction on $t \in [t_1, t_2]$, that the registry for k is up to date at t:

— if $t = t_1$, the claim holds by assumption.

— if $t = p.t + \tau$ for some put operation on k, then the registry is up to date for k at t by Lemma 1.

— otherwise, we apply induction hypothesis, which gives us an up-to-date registry at time $t-1$, and we apply Lemma 2. No operation was ongoing on k at $t-1$, otherwise we would be in the previous case, so we can apply the induction hypothesis.

This holds for any k, as soon as no put on k is ongoing at t. Therefore, if no operation is ongoing overall, this holds for any k, and the Lemma holds. □

Lemma 4. *Let k be a key and t be a time. n stable state, if no put operation is ongoing on k, ρ_k always returns a node id that contains latest(k, t), the latest data.*

Proof. According to the system properties under stable state (S1), there are two cases with respect to the properties 3.1 and 3.2:

1. $\rho_k(k)$ is defined. As it is up to date, it returns some $r \in R_l(k)$.
2. if the entry for k *does not exist* in ρ_k, then ρ_k returns the id of any $r \in h(k)$. According to Property 3.2, $\rho_k(k)$ undefined implies $R_l(k) = h(k)$. ρ_k returns any $r \in R_l(k) = h(k)$, but all the replica nodes contain the latest version.

□

Theorem 1. *Let k be a key. If there is no ongoing write operation on k, then any user reads the most recent version of k.*

Proof. From Lemma 4, ρ_k always returns a node id (IP address) that contains $latest(k, t)$, when in stable state. According to Lemma 3, the system is in stable state, except temporarily, when a write operation is ongoing. So users read the most recent version of k. □

3.3.4 Unstable States Properties

When in an unstable state, information returned by the registry can be wrong in two ways:

— The registry does not contain a subset of $R_l(k)$, which means ρ_k is not up to date for k:

$$\rho_k \nsubseteq R_l(k)$$

The probability that the registry returns an id (IP address) of a node that does *not* contains *latest(d, t)* is $1 - \frac{|common|}{|\rho_k(k)|}$, where $common = \rho_k(k) \cap R_l(k)$.

— The registry contains no entry for k. The probability that the registry returns the id of a node that does *not* contains *latest(d, t)* is $1 - \frac{|R_l(k)|}{|h(k)|}$.

Lemma 5. *Let r be a node with registry ρ, and $k \in dom(\rho)$. When r leaves the ring, the registry entry $\rho(k)$ will be rebuilt on the new registry node ρ_k after a successful write operation on k.*

Proof. In the event of node joining or leaving the cluster (including nodes failure/ disconnection), the underlying system guarantees helps to detect it at the earliest. In such a case, there will be a change in the first available replica node as mentioned in the section 3.3.2.2. As soon as this change has been made, and assuming that no node joins or leaves the ring, we are in the conditions of Lemma 1, that guarantees that the registry ρ_k is up to date for k after the write operation, and this property is hereditary (Lemma 2), so up-to-dateness for k holds back (see the proof of Lemma 3), as long as all put operations on k are successful, and there is no new change in the ring membership. □

In particular we might encounter some $g(k,t)$, with $t \geq t_0$ (t_0 is the failure time of r), for which ρ_k returns $r \notin R_l(k)$. However, as we just saw, after the first $p(k,v,t')$, such that $t' > t_0$, $\rho_k(k)$ is up to date again and preserves this property. Therefore, for all $g(k,t'')$ with $t'' \geq t' + \tau$, ρ_k returns $r \in R_l(k) \subseteq h(k)$.

Since the put operation following the stale read updates the registry, for all subsequent get operation, ρ_k returns an id (IP address) of a node that contains the $latest(k,t)$.

Now, assume that r leaves (or joins with a stale registry, or gets a stale registry due to a write failure, for the purpose) the ring, with registry ρ, at t_0. Assume also, that a write operation happens on all the data $k \in sup(\rho)$ (the keys, of which node r has the charge), such that the new ρ_k was *not* up to date. The registry entry $\rho_k(k)$ will be rebuilt on the new registry node and is now up to date. For the data $k \in sup(\rho)$, such that ρ_k (the new registry) was up to date from the inception (see discussion in Section 3.3.2.2), there is no need for rebuilding, so no need for a write operation. As a consequence, all registries are up to date and the system has switched back to the stable state S1. This yields the following theorem.

Theorem 2. *Assuming no further joining/leaving node event or write problem, the time taken by the LibRe protocol to recover from an unstable state is the minimum among the time for the next successful write operation on stale data T_{st} and the time taken to resolve the staleness via underlying system's eventual consistency guarantees P_{st}.*
From any unstable state, we converge towards stable state S1 in time $\min(T_{st}, P_{st})$.

Proof. Let the time that a quorum based replication system resolves staleness by itself via read-repair and generic anti-entropy protocols is given by P_{st}.

From Lemma 5, the registry for the stale data is rebuilt in the new nodes after the next write operation, and we need one write operation per stale data (T_{st}).

Either of those solutions ensures that all the registries are up to date. From the above two propositions, the time taken by the LibRe protocol to recover from the unstable state is $\min(P_{st}, T_{st})$. □

We have shown different states of the LibRe protocol $S1, S2, S3, S4, S5$ and the properties of the protocol under each state. During normal condition, when there is no change in the ring membership and no write failure, the protocol remains in state $S1$. LibRe under state $S1$ ensures consistency of the read operations and we call it as *stable state*.

When a node joins or leaves the ring, change in the ring membership affects the consistency guarantees of the protocol. The event of a node leaving (unreachable to) the ring is denoted as state $S2$ and the event of node joining (or recovering from disconnection) the ring is denoted as state $S3$. Since the system exhibits the same property under both states $S2$ and $S3$, the two states are denoted commonly as *unstable state*, along with other states resulting from failure of a write operation. LibRe under unstable state ($S2$ or $S3$) do not ensure consistency of the read operations. We also formally showed that protocol under unstable state will switch back to the stable state $S1$ within short period of time.

3.4 Related Works

In the literature, there are multiple works aiming at improving the performance and reliability of quorum systems [MRW97; MR97; AEA]. However, in all these works, a suf-

ficient amount of nodes has to be contacted in order to satisfy the intersection property. Apart from the works on quorum systems, there are also a few works in the literature whose approaches are similar to some of the techniques followed in the LibRe protocol. One of the most famous work that has similar approach of the LibRe protocol is the 'NameNode' of the Hadoop Distributed File System (HDFS) [SKRC10].

The HDFS NameNode manages metadata of files in the file system and helps to locate needed data in the cluster. But, on the contrary to the LibRe registry, which maintains metadata about small data items, the HDFS NameNode manages metadata of large file blocks. In addition to this, the NameNode is a centralized registry that stores information about the whole cluster, which can make the whole system unavailable in case of failure of the NameNode. In our approach, the LibRe registry only stores the location of partially propagated writes, and in case of failure of a registry node, availability of the system is not affected.

BigTable [CDG+08], which is a data store designed by Google uses a two level lookup before contacting the actual data node for accomplishing reads and writes. In BigTable, the UserTable that needs to be contacted for accomplishing reads and writes is found by looking at a ROOT tablet followed by a METADATA tablet. This enables to have a scalable and fast lookup. The earlier version of HBase [Geo11], which is an open source implementation of BigTable, used a similar two-level lookup for finding data in the system. In the later version, the two-level lookup is reduced to a single lookup in the METADATA table. However, BigTable and HBase ensure strong consistency, so there is no context of stale replicas in these data stores. In LibRe, we use a single lookup to identify a fresh replica node only for reading some data items that are configured to use LibRe protocol.

In [LAS13], Liu et al. describe a storage service called DAX that follows a similar approach as LibRe for avoiding stale reads from an eventually consistent data store. The DAX system uses a similar in-memory registry to keep track of the most recent version of each data item. But unlike LibRe, where the registry is maintained at the database side, in DAX the registry is maintained at the client side. During read operations, the client refers the local registry to make sure the version read from the database is not stale. If a stale read is detected, the client retries the read operation with a stronger consistency

option.

In [TAPV10], Tlili et al. designed a reconciliation protocol for collaborative text editing over a peer to peer network using a Distributed Hash Table (DHT). According to the protocol, for each document, a master peer is assigned via the lookup service of the DHT. The master peer holds the last modification timestamp of the documents in order to identify missing updates of a replica peer in order to avoid update conflicts. This master-peer assignment is similar to the *Registry Node* assignment in the LibRe protocol. However, in LibRe, the registry node holds the version-id of the recent update and the replica nodes holding it in order to avoid reading from a stale replica.

The Global Sequence Protocol (GSP) designed by Burckhardt et al. in [BLPF15] uses two states for an update such as known sequence and a pending updates sequence. When an update is issued by a client, the update is kept in the pending updates list, and broadcasts its origin and a sequence number to all the replicas. Once an echo is received confirming that all the replica copies received the update, the particular update from the pending updates list is removed. Similarly, in LibRe, when an update is applied on a replica copy, the version-id of the update along with the replica id is kept in a *Registry*. Once a confirmation is received from all the replica nodes, the entry for the corresponding data item will be removed from the registry. However, GSP focuses on ordering the write operations, whereas LibRe focuses on reading the value of the recent write.

The PNUTS Database [CRS+08] from Yahoo uses a per-record mastership over per-table or per-tablet mastership and forwards all updates of the record to this master in order to provide timeline consistency during read operations. In contrast, LibRe allows any replica to process an update and chooses a registry node per data item (per-record mastership) in order to identify the most recent version of the data item.

3.5 Summary

The LibRe protocol described in this chapter aims at enhancing the tradeoffs between Consistency, Latency and Availability of an eventually consistent Key-Value store. Our

protocol: LibRe prevents the system from forwarding read requests to the replica nodes that contain stale version of the needed data item. In order to identify replica nodes that contain stale replicas, LibRe maintains an in-memory data structure (registry) that keeps track of a list of data items whose recent version is not synchronized to all the replica nodes. The write operations under LibRe protocol issues a success message to the client as soon as a write operation is successfully accomplished on one of the replica nodes. After successful write operation, each replica node updates the in-memory registry using the version-id of the recent update. The monotonically increasing guarantee of the version-id helps to maintain the in-memory registry with the IP-addresses of replica nodes that hold the recent version of each data item. Referring the registry during read operations helps to forward the read requests to the replica nodes that hold recent version of the needed data items, by the way avoids stale reads.

When there is no change in the ring membership, using LibRe protocol helps to read the most recent version of the needed data items with minimum read and write guarantees. It is denoted as *stable state* of the protocol. However, when there is churn in the system, change in the ring membership affects the consistency guarantees of the read operations. And it is denoted as *unstable state* of the protocol. The protocol relies on the underlying system guarantees for recovering back to stable state. We gave a formal description of the behavior of the LibRe protocol under stable and unstable state and the guarantees that the protocol under unstable state will eventually converge back to the stable state. Due to these two different states of the protocol, LibRe can be considered as an intermediary consistency option that lies between eventual consistency and strong consistency guarantees of the quorum-based replication systems. In the coming chapter, we will discuss performance evaluation of LibRe against popular consistency protocols via simulations followed by the performance results of the prototype implementation of LibRe inside Cassandra.

Chapter 4

Evaluation of Consistency Protocols via Simulations

Due to CAP theorem, most of the distributed database systems rely on eventual consistency and do not offer strong consistency by default. In order to offer better performance with acceptable level of consistency, these storage systems offer different consistency options. These consistency options enable the user to choose needed tradeoff between Consistency, Latency and Availability of the system. Hence, in order to reason on an appropriate consistency option for different workload patterns rigorous benchmarking are demanded. Benchmarking the different tradeoffs between Consistency, Latency and Availability remains challenging both via simulation as well as in a real-world system with a live workload.

Distributed data simulations tools were initially focused on grid architecture. These tools were later adapted for cloud architectures by adding extra features over grid simulators like support for virtualization. Currently, the ubiquitous approach to simulate a new policy/strategy for distributed computing architecture is to use one of these existing simulation libraries. Some of the popular libraries among those are SimGrid [BLM+12], OptorSim [BCC+03], GridSim [MB02], CloudSim [CRB+11b]. These utilities allow simulating a cloud data center infrastructure by using a dedicated JAVA or C/C++ API defined by the library. Each of the simulators deals with diverse features offering larger scopes for environment level simulations.

CloudSim [CRB+11b], which is one of the popular simulation toolkit has provision for

almost all types of simulations. Some of the supported simulations in cloudsim are modeling Data Centers, Virtual Machines, Network behaviors, Power Consumption, efficiency of dynamic load balancing strategies on the overall platform performance. As we can see, these simulations are mainly focused on environmental level behaviors rather than application level behaviors.

OptorSim [BCC+03], which has emerged as part of the European DataGrid project [1] focuses mainly on Cost estimation based on file replication strategies for different topological setup and workloads. The Simgrid Utility [BLM+12] was originally designed for transparently simulating massive distributed computations before their effective deployment on a grid infrastructure. Another project called the GreenCloud Simulator, which relies on the *NS-2* network simulator [MFF] aims at modeling the energy consumption of Cloud Computing data centers. Last but not least, the iCanCloud [NVPC+12] project uses a mathematical model to evaluate the costs of using a Cloud Computing platform such as the Amazon Elastic Compute Cloud Platform (EC2) [2].

As it appears, these simulators are aimed more on simulating the underlying physical infrastructure. They give useful information to the Cloud Computing service providers, regarding cost or efficiency against a specific type of workload. They do not allow the user to study the impact of some application level protocols such as load balancing strategies, consistency policies or the combination of both. By the shared and multi-tenant nature of Cloud infrastructures, application performance prediction of cloud based applications remains difficult [BS10; IOY+11]. Therefore, there is an existing need to provide this kind of simulation tool. Also noted in [SL13a], to this date, there is no cloud infrastructure simulator that allows to assess the behavior of an application deployed in such an environment. Some efforts are made in this direction, such as the CDOSim simulator [FFH12], but they are not publicly available so far. Hence in order to provide a simple programming interface for modeling distributed applications and data storage systems, we developed a new simulation tool called Simizer [LKC14].

In this chapter, we describe how to evaluate the performance of a consistency protocol

1. `http://eu-datagrid.web.cern.ch/eu-datagrid/`
2. `http://aws.amazon.com`, October, 5th, 2013

against different consistency protocols using Simizer. The detailed information about the internals and the correctness of the working principles of simizer are described in [Lef13]. This chapter is organized as follows: The general architecture description of simizer is described in the Section 4.1. The fundamentals about describing a workload and server configuration are discussed in the Section 4.2. Section 4.3 describes how to simulate different consistency options using simizer with examples. The simulation results of LibRe performance against different consistency protocols are discussed in Section 4.4 followed by concluding remarks.

4.1 Simizer description and architecture

Simizer is a discrete event simulator written in JAVA language. It is based on a three-layer architecture, presented in figure 4.1. The bottom layer is the Event layer, which provides event management utility classes and random number generation. The Application layer is used to implement application level protocols. The Architecture Layer uses the event layer classes to model virtual machines and networks behavior. The Architecture Layer of Simizer represents the different entities subject to simulation. In the case of Simizer, these systems are Virtual Machines executing a set of communicating applications. Simulated entities are represented in figure 4.2.

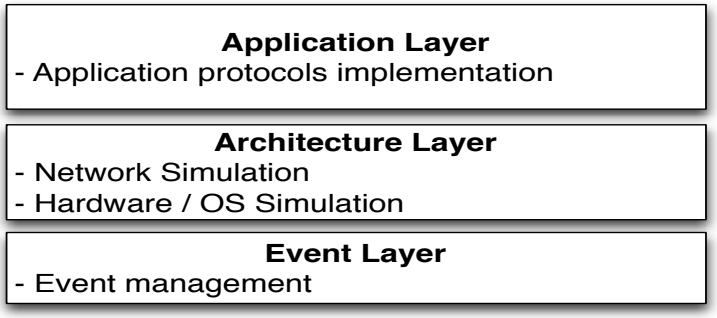


FIGURE 4.1 – Architecture de Simizer

4.1.1 Entities

The Network Entity models the behavior of LAN or WAN networks. It is possible to use pluggable networks model such as those found in other simulators such as [BLM+12]

and adapt it to Simizer. As for the time being, the current model of network simulation relies on Gaussian distributed transfer latency. The values used for latency in LANs and WANs are those of one presented in [Sco12]. Nodes are the entities that can communicate with each other through a Network. There are three kinds of Nodes:

ClientNode: This class models users' behaviors via different models that can be derived and modified.

ServerNode: A ServerNode instance executes Requests received from the Client Nodes. These entities model the application behaviors.

LBNode: LBNode models an application level load-balancer in order to evaluate different policies to distribute requests to the different ServerNodes of the system. It sits as an intermediary between ClientNodes and ServerNodes.

Clients generation can be controlled by the ClientManager class, which commands new ClientsNodes creation following a user configurable random distribution. When a ClientNode is created, it sends a new Request through the associated Network object. The Network will apply its model to raise an event of RequestReceptionEvent when the request reaches its destination ServerNode or LBNode. During this time, the Request is encapsulated in a Message object indicating the destination Node (cf. figure 4.2). A Request is composed of a set of Resources, a list of parameters and the number of Instructions needed to process the request. Resources are objects stored on the machines disks and therefore represent chunks of data that can be read, written and modified during a request execution. The number of instructions defines the amount of processing power needed in order to complete the Request execution. Once the request is completed, the ServerNode executing it sends a response to the ClientNode. On each ServerNode, a Processor simulates task execution. A processor is defined by its number of cores and the amount of instructions that each core can execute in one second in MIPS (Millions of Instructions Per Second). Disk storage and Random Access Memory (Cache), are respectively modeled by the StorageElement and Cache classes.

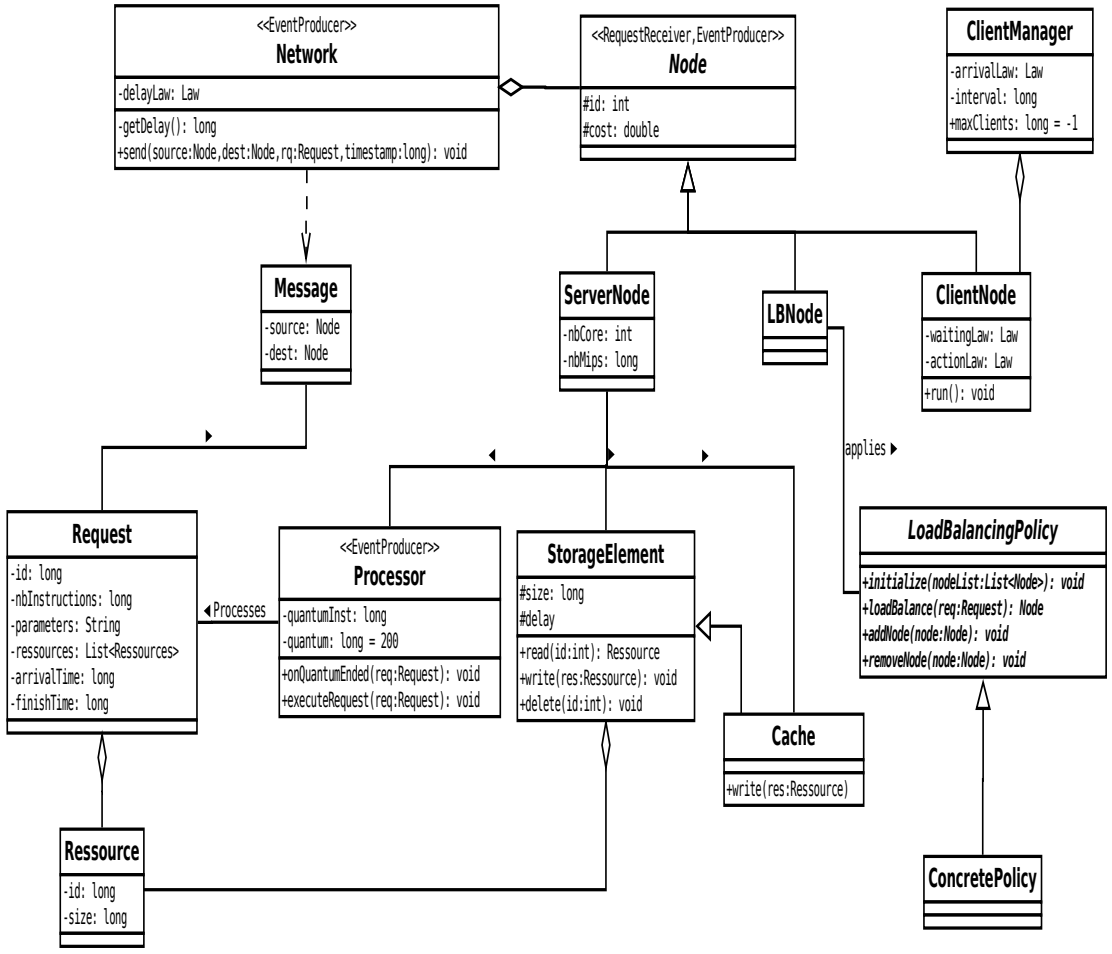


FIGURE 4.2 – Architecture simulation classes

4.1.2 Processor simulation

The Processor class evaluates the latency involved in processing a particular read or write request. The latency is evaluated based on the amount of instructions needed to process the requests in addition to application-specific constraints. The application-specific constraints include the time taken to retrieve a particular data (from cache or disk) and locks obtained on a particular data. The processor can switch tasks while associating the time taken to retrieve a particular data from the disk or until lock acquired on a particular data is released. Task switching is based on the definition of so-called *quantum* of time. A quantum is an amount of time during which the processor executes a single task. At the

end of the quantum, the processor changes the currently executing task for another one in its task list (in Simizer it is a Request list), depending on the operating system task scheduling policy. In current Linux-based operating systems, this value is usually set to 4 milliseconds by default [Mol07]. Each quantum end is signaled to the processor through an event, and the next task will be started from the list of queued tasks. Tasks in the list are ordered by quantity of instructions left to execute, which entails faster execution of smaller tasks, which is a common approach in operating system scheduling. For example, a single task of 10000 MI submitted to a machine equipped with a processor holding 1000 MIPS of processing power should provide an overall simulated execution time of 10 seconds.

4.1.3 Request Execution

When a Request arrives to its destination, an event is triggered on the targeted ServerNode. The ServerNode then passes the request to its Processor for execution. Before executing the request, the processor must load the request resources in the main memory. The delay incurred by this action is calculated according to the size of the resources and the speed of the ServerNode disks (StorageElement). When the resources are loaded, the processor starts processing the instructions of the request. When the request execution is over, processor will signal a RequestEnded event to the executing ServerNode, which then sends back a response to the ClientNode through the network. Upon response reception, the client prints out the result as the request description, and the amount of time it took to execute this particular request.

4.2 Simulator usage

Simizer can work in three different modes: a *trace generation mode*, a *trace replay mode*, and a *generative mode*. The *trace generation mode* enables the user to generate a request sequence based on a set of random distributions specified in a configuration file. Default laws are described in table 4.1. The result of this generation is stored in a Comma Separated Value (CSV) file, called the workload description file. Each request is stamped by a timestamp (the Request Arrival Time) indicating the time at which the request has to be sent to the servers. This file can also be written manually, and is useful to test a

TABLE 4.1 – Distribution laws available in Simizer

law	parameters
Exponential	mean (λ)
Gaussian	mean & standard deviation (μ, sd)
Poisson	mean (λ)
Uniform	density
Zipf	skew (s)

specific request sequence that would cause problem to the system.

Using this workload description file, it is possible to launch a simulation by specifying the number of servers required in the system and by choosing a request distribution strategy. This is the *trace replay mode*, where the simulator will send the requests following the order specified in the CSV file.

The *generative mode* is the third execution mode of Simizer, and consists in direct simulation of both the clients and servers behaviors. In this mode, clients are dynamically created and removed according to some user specified laws and the simulator directly computes requests execution times for the generated workload. The user can specify the quantity and characteristics of servers through a specific configuration file.

In these three modes, three main configuration files have to be edited by the user: the *workload description file*, which specifies clients behavior, the *request description file* that provides the models of requests that will be sent by the clients and the *server description file* specifying the characteristics of the simulated servers.

4.2.1 Workload description files

The different distribution laws used to determine the clients behavior are specified in a configuration file that is passed to the simulator and is called the 'workload definition file'. These laws govern four aspects of the load as described below.

Request selection Law: Every client in the simulated system is an independent agent, which can send a certain number of requests to the servers. For each new request, the client selects a set of parameters among a predefined list. This selection can be done according to a specific law to model some popular user behaviors [Fei02; GJTP12].

Client arrival law: Specifies the client arrival process and is defined by a time interval and the average number of new clients being created during this interval.

Client lifetime law: Each client is created for a given amount of time. Client lifespan is usually modeled by an exponential law [Fei02; GJTP12].

Client think time law: After receiving a response for a request, a client may send a new request to the servers. The amount of time between the response reception and a new request is called the 'think time' and can be modeled by some probability laws.

Each of the laws described in table 4.1 can be used to define one of these four descriptive parameters. As the Law classes are dynamically loaded, it is possible for a user to define his/her own laws and add it to the simulator.

4.2.2 Request description file

In addition to the workload description file, simizer requires a description of the requests to be sent to the simulated infrastructure. This description specifies the list of resources accessed by each type of request, and the quantity of instructions to be processed in order to achieve the request. The simulation generates the actual requests issued by the clients from these descriptions. A request can be of different types, depending on the kind of actions that it has to carry out. These basic types are the Read and Write operations. In a Read request, the specified resources are read from the disk before starting execution of the instruction set. In a write request, data is written to disk after the instructions are carried out. At the current state of development, these kinds of requests are enough to simulate the use cases presented in this work. However, future works will enable users to implement user-defined semantics based on these basic operations.

4.2.3 Servers description file

The server description file is a JSON formatted file describing the capacity of the different servers that the user wants to use in his test. This file describes the hardware characteristics of a given machine such as the disk size, the type of processor, the number

of cores and processing capacity of its processor, the size of the RAM, and the hourly cost of the machine.

Accordingly, the user can specify the number of servers he wants of each kind. The example file displayed in figure 4.3 describes an infrastructure composed of 4 machines, with two machines having 1GB of RAM, 512GB of disk storage, and two cores each capable of processing 1500 MIPS, and another set of two machines having twice these capacities.

```
[
        {
        "nb":2,
        "memorySize": 1048576,
        "diskSize":512000000,
        "cost":10,
        "ProcessorName":
        "simizer.processor.LinuxProcessor",
        "cpuSlots":2,
        "nbMips":1500.0,
        }
        {
        "nb":2,
        "memorySize": 2097152   ,
        "diskSize":1024000000,
        "cost":15,
        "ProcessorName":
        "simizer.processor.LinuxProcessor",
        "cpuSlots":4,
        "nbMips":1200.0,
        }
]
```

FIGURE 4.3 – Server description file example

4.3 Simulation of Consistency options with Simizer

In this section, we discuss the pseudocode of an optimistic consistency protocol followed by LibRe and how these pseudocodes mimic the appropriate consistency protocols in accordance to the performance results. The Simizer application layer provides two interfaces for implementing request distributions strategies, and/or consistency policy on the data accessed by the requests.

The main metrics in the evaluation of consistency-latency tradeoff are the number of stale reads and the average execution time of the requests. A stale read is defined as

reading a resource from a ServerNode that is not up to date compared to other versions of the same resource stored in other ServerNodes. As summarized by the CAP theorem, after a certain scale, consistency in a distributed system can only be maintained at the expense of system availability or partition tolerance. Further work on this topic showed that availability of the system is closely coupled with the latency of the request [Aba12]. Therefore, measuring the number of stale reads and comparing it to the response time of the system is an adequate way to evaluate the efficiency of any consistency protocol.

The implementation of a consistency protocol relies on the resource replication (aka. data replication) policy. In order to account on the replication delays, each resource on a given ServerNode is associated to a so-called *alive time*. It is the time after which a resource is available to the machine storing it. The consistency policy will determine how the resources are replicated in the system by calculating and setting the appropriate alive time for each resource. Hence, a stale read is detected either when a specified resource is not found on the ServerNode processing the current request or when the associated alive time of the resource is not yet passed. The later case means that the resource is not currently consistent with its replicas, which are stored on other nodes. In order to avoid reading a resource with an advanced alive time, one can use locks to block access to this resource until the alive time matches the simulation clock time.

To implement such a protocol in Simizer, a user needs to derive the Processor class and implement three operations: the Read, Write and Modify methods. These methods enforce access rules to data on each node and can use different techniques to enforce data synchronization. This enables the implementation and comparison of several consistency protocols and helps to obtain acceptable results when compared to the theoretical behavior of the protocols.

As an example, listing 4.1 shows the implementation of read and write operations of an optimistic replication (eventually consistent) policy [Vog09]. In this strategy, newly written data is replicated to all machines in the system, with an alive time corresponding to the data local write time added to a fixed probabilistic replication time (line 5 to 16). Since the new data is replicated without any locks, this policy is known as optimistic replication policy [SS05]. Verifying the alive time associated with a particular data ensures whether

a read operation returns a stale value or not.

Listing 4.1 – Optimistic consistency Protocol

```
1
2  public class OptimisticPolicy implements Processor {
3
4  public Request write(Request r) {
5      aliveTime = systemTime;
6      // local write
7      nodeInstance.getStorageElement().write(r.getResources(),aliveTime);
8      nodeInstance.getCache().write(r.getResources(),aliveTime);
9
10     // Replication to other nodes
11     for(Node n:nodes) {
12
13         if(n.getId() != nodeInstance.getId()) {
14               aliveTime += gossipTime;
15           n.getStorageElement().write(r.getRessources(),aliveTime);
16           n.getCache().write(r.getRessources(), aliveTime);
17           }
18      }
19 }
20
21 public Request read(Request r) {
22     StorageElement disk = nodeInstance.getStorageElement();
23     Cache cache = nodeInstance.getCache();
24     for(Integer rId: r.getRessources()) {
25
26          if(cache.contains(rId)) {
27               Ressource res = cache.read(rId);
28
29                         // check alive time
30               if(res.getAliveTime() > gSystemTime){
31                 // data is not valid to be in cache yet
32                   accessTime += disk.getDelay(rId);
33                   r.setError(r.getError()+1);
34               }
35
36          }
37          else if(disk.contains(rId)) {
38                 // reading from disk is slower
39            accessTime += disk.getDelay(rId);
40            Ressource res = disk.read(rId);
41
42            if(gSystemTime < res.getAliveTime())
43                 r.setError(r.getError()+1);
44                //cache update
45                if(res != null)
46                    cache.writeToCache(res);
47         }
48         else { // if data does not exist in cache as well as disk.
49                  accessTime += disk.getDelay(rId);
50                  r.setError(r.getError()+1);
51           }
```

```
52      }
53  }
54  }
```

In case of a read request, the data needed for processing the request will be fetched from the appropriate nodes. As shown in listing 4.1, while fetching the data, the node checks whether the needed data exists in the node's cache (line 26 to 33). If the data is found there, then the Resource Access Time (cf. section 4.1.3) will be taken as zero, else the Resource Access Time will be calculated (lines 32, 39, 49) based on its size to retrieve the data from its disk [Sco12]. If the needed resource does not exist in the disk or the alive time of the data is greater than the current system time, an error bit will be set on the request indicating a stale read (lines 33, 43, 50). After a request is processed, the latency of the request and an error count (if any) along with the request will be sent back to the front-end node (LBNode).

Listing 4.2 – LibRe consistency protocol

```
1  public class LibrePolicy implements Processor {
2
3  public Request write(Request r) {
4      aliveTime = systemTime;
5      // local write
6      nodeInstance.getStorageElement().write(r.getResources(),aliveTime);
7      nodeInstance.getCache().write(r.getResources(),aliveTime);
8      LibReService.insert(r.getRessource(), node_id, aliveTime, version_id);
9
10     // Replication to other nodes
11     for(Node n:nodes) {
12
13         if(n.getId()!=nodeInstance.getId()) {
14               aliveTime += gossipTime;
15           n.getStorageElement().write(r.getRessources(),aliveTime);
16           n.getCache().write(r.getRessources(), aliveTime);
17           LibReService.insert(r.getRessource(), n, aliveTime, version_id);
18           }
19      }
20 }
21
22 public Request read(Request r) {
23     StorageElement disk = nodeInstance.getStorageElement();
24     Cache cache = nodeInstance.getCache();
25     for(Integer rId: r.getRessources()) {
26
27          if(cache.contains(rId)) {
28               Ressource res = cache.read(rId);
29
30                         // check alive time
31               if(gSystemTime < res.getAliveTime()){
```

```
32                      accessTime+= disk.getDelay(rId);
33                      r.setError(r.getError()+1);
34                  }
35
36              } else if(disk.contains(rId)) {
37                      // read from disk is slower
38                  accessTime+= disk.getDelay(rId);
39                  Ressource res = disk.read(rId);
40
41                  if(gSystemTime < res.getAliveTime())
42                      r.setError(r.getError()+1);
43                     //cache update
44                     if(res != null)
45                         cache.writeToCache(res);
46              }  else{ // error in that case.
47                      accessTime+= disk.getDelay(rId);
48                      r.setError(r.getError()+1);
49               }
50      }
51  }
52  }
```

The listing 4.2 shows a simulation code example of the LibRe protocol. The only difference between the LibRe protocol and optimistic replication protocols showed in listing 4.2 and 4.1 respectively is that with LibRe protocol, the application sends an extra notification message to the LibRe service during write operations (lines 8, 17). Since the frontend forwards the read requests to a right node by consulting the LibRe service, the reads in LibRe protocol will always see the most recent version of the data without any additional modifications to the read path of the system. Hence, the read path of the application is the same as that of the optimistic replication protocol shown in listing 4.1. However, at the request reception side, the frontend node has to fetch the right replica node id from the LibRe service while forwarding the read requests.

4.4 Consistency Evaluation using Simizer

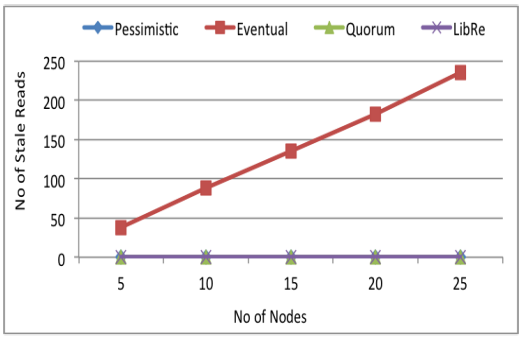
In this section we present the performance results of LibRe protocol against popular consistency options of quorum based replication systems such as: Read One Write All, Write Quorum Read Quorum, Read One Write One.

Test Setup

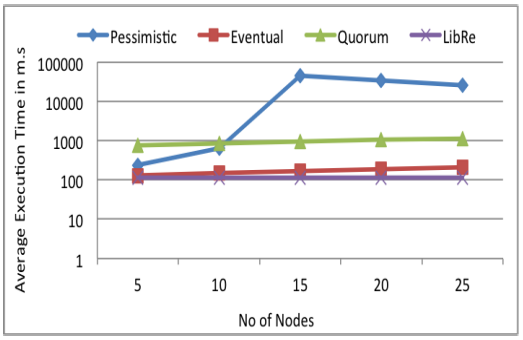
In our test setup, we consider 1200 requests, out of which 400 of them are write operations and the rest are read operations shuffled in random order. During write operations, data will be written in nodes cache and disk. During read operations, the latency of the requests is calculated based on the number of instructions needed to process the request in addition to access time of the needed data. If the needed data are available in the node's cache then resource access time is taken as zero, otherwise the resource access time will be calculated based on the size of the data. If a needed data does not exist, an error flag will be set in the request indicating a failure. The tradeoff between consistency and latency is performed with the same set of requests handled through different consistency policies such as pessimistic protocol (Synchronous Replication), eventually consistent protocol (Asynchronous Replication), Quorum replication and LibRe increasing the number of nodes from 5, 10, 15, 20, and 25.

Test Result

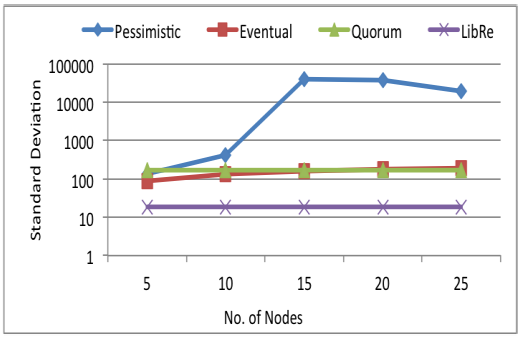
Figure 4.4 shows the obtained test results. From figure 4.4a, we can see that except eventual consistency policy all other policies offer tight consistency. But when we look at the request latency, we can see a considerable delay in the request latencies of Quorum and Pessimistic protocol. We can also see that the average request latency (figure 4.4b) and standard deviations (figure 4.4c) are lesser in LibRe than in other policies and are not affected by increasing the scale. LibRe behaves that way because requests are forwarded to the node where the write operations have just been performed for the same data items. Therefore, the data items will be in its cache and hence the resource access time will be low. Both eventual consistency and pessimistic protocol take advantage of the cache memory. In the case of eventual consistency, since requests are forwarded without the account on cache memory optimization, some of the requests are benefited from the cache memory while some are not. In the case of the pessimistic protocol, almost all the requests are benefited from cache memory, but it was affected by the waiting time of the requests until the locks on certain data items are released while doing synchronous writes. Hence, the request latency is highly variable for the pessimistic approach and more predictable for



(a) Nb of Stale Reads out of 800



(b) Average Total execution Time in m.s



(c) Standard deviation

FIGURE 4.4 – Experimental Results

other policies which can be seen clearly in the standard deviation graph (figure 4.4c). In the case of quorum, since the requests are forwarded to a majority of nodes in the cluster, the possibility that all nodes get the right data in their cache is mostly limited.

4.5 Conclusion

Replicated data consistency is closely related to the application-specific constraints. Hence, in order to evaluate consistency protocols, it is important to take into account the application related constraints. Existing cloud based simulation tools do not provide necessary functionality to simulate consistency policies or other application-specific behaviors. Simizer helps to simulate applications in order to evaluate different consistency policies along with the application level metrics. Simizer is an event-based simulation library that enables fast and efficient simulation of distributed computing infrastructures. By using simizer, different consistency and data synchronization strategies can be tested

in an efficient way before evaluating the performance of a protocol in a real cloud based distributed computing system. Simizer provides an intuitive way to describe workload and infrastructure characteristics, enabling the user to concentrate only on the protocol or strategy needs to be evaluated. The simulation results of simizer provide precise and detailed information that allow users to evaluate the simulated protocols on different aspects such as latency, availability and scalability of the system. The description about how to simulate a consistency option and its performance results are described with the code. The performance of the proposed consistency protocol LibRe was tested against some of the popular consistency protocols. With the encouraging performance results of the LibRe protocol, the next chapter will discuss more about the prototype implementation of the enhanced version of the LibRe protocol inside Cassandra.

Chapter 5

Performance of LibRe against Cassandra's Native Consistency options

In this chapter, we discuss the implementation of the enhanced version of LibRe protocol that is described in Chapter 3 inside the Cassandra distributed database system. The implemented version of LibRe protocol adapted to the internals of Cassandra is named CaLibRe, which signifies *Cassandra with LibRe*. As stated in Chapter 4, evaluating different tradeoffs of a consistency policy remains challenging both via simulation as well as with a real system. In order to evaluate the performance of different consistency options provided by a cloud database, we extended an existing cloud database benchmark tool YCSB (Yahoo! Cloud Serving Benchmark) to provide an additional metric regarding data consistency. This additional consistency metric along with the default latency and throughput metrics offered by YCSB help to evaluate consistency-latency tradeoff of different consistency options. This chapter is organized as follows: Section 5.1 describes the implementation of LibRe protocol inside Cassandra (CaLibRe). The extension to YCSB benchmark tool that is used to evaluate the consistency metrics of Cassandra is described in Section 5.2. Performance results of CaLibRe against Cassandra's native consistency options are discussed in Section 5.3 followed by conclusion and future works.

5.1 Implementation of LibRe inside Cassandra

Cassandra [LM10] is one of the most popular open-source NoSql systems that satisfies the target system model specified in section 3.2.1 of Chapter 3. Hence, we decided to implement the LibRe protocol inside the Cassandra workflow and evaluate its performance against Cassandra's native consistency options: *one*, *quorum* and *all* [Hew10]. Although Cassandra is a column family data store, we used it as a Key-Value store during test setup (refer Section 5.3.1).

5.1.1 Cassandra data model

Cassandra design is refined from two popular NoSql systems BigTable [CDG+06] and Dynamo [DHJ+07b], borrowing the data access model from BigTable and data partitioning scheme from Dynamo. Like BigTable, Cassandra identifies each data by its row key and Column key. Data are first logged to an in-memory table called 'MemTable', which will be compacted to SSTable, then the set of SSTable together will be considered as a table. Like Dynamo, the data-partitioning scheme of Cassandra organizes servers in a ring structure and replicates the data among the neighboring nodes.

5.1.1.1 Ring Structure

In the initial versions of Cassandra, each node in the cluster is a physical node organized in a ring like structure that ranges from 0 to $2^{127 - 1}$. However, the recent versions of Cassandra releases follow the virtual nodes technique similar to Dynamo. A token number identifies each node (virtual node) uniquely. These token numbers are assigned depending on the total number of nodes in the cluster and taking into account the node's capacity. The token numbers represent the nodes position in the ring. Each node is responsible for the data whose token numbers are less than or equal to the node's token number. The node which is responsible for including other nodes into the ring by assigning token numbers is called Seed Node. There can be more than one seed node to avoid any potential failure. The seed node addresses are to be specified in the configuration files of each node in the cluster. But at least one of the seed nodes should be up to join the new node in

the ring. When a new node is authenticated to join the ring by any of the seed nodes, the information will be propagated to all the nodes in the ring. Each node in the ring stores all the information about the ring locally including the data range for each node.

5.1.1.2 Data Model

Cassandra is a column-family data store, which is a variant of key-value data store. Unlike holding a single value for a key as in key-value data store, Column-family data store can hold more than one value each representing a column in a column-family. In column-family data store, each data is identified by a unique row-key. The row-key will identify the column-family to look-up for the needed column, and the name of the column inside a column family represents the actual data. In other words, each data will be retrieved by the combination of the row-key, the column-family name and the name of the column to retrieve [SF12].

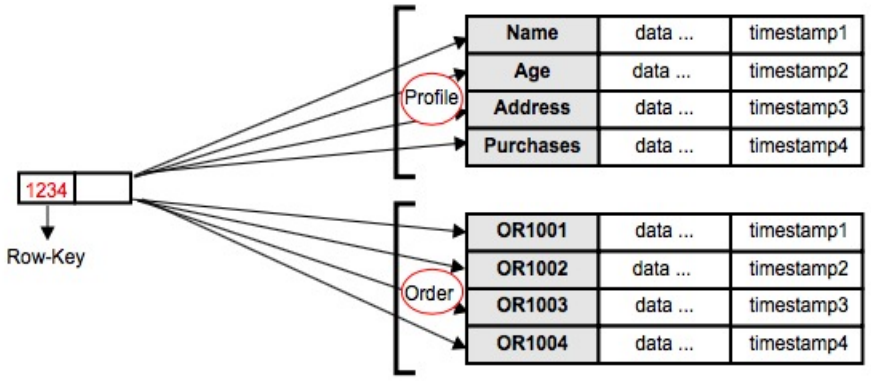


FIGURE 5.1 – Column-Family data store

Figure 5.1 describes a column-family storage model. The figure depicts a classic shopping cart example, in which the actual data is identified by the row-key (1234), and the column-family name (profile/orders). For instance, the combination of the row-key '1234' and the column-family name 'Profile' can identify the columns such as Name, Age, Address, Purchases. In the similar way, the combination with column-family name 'Orders' can identify the columns OR1001, OR1002, OR1003, OR1004. The main idea of the

column-family data store is to store related set of data (columns) that are frequently accessed by the application together in order to benefit the read performance [SF12]. Each column in a column-family data store is a triplet, containing three fields: column-name, column-value and timestamp. The column-name represents the name of a data to retrieve, the column-value is the actual value for the data, and the timestamp is used to identify the recent version of the data in case of update conflicts.

5.1.1.3 Read-Write Path

As Cassandra is a peer-to-peer system, a client can connect to any node in the cluster to query a data. The node to which a client connects to query a data is called Coordinator node. The coordinator node is responsible for identifying the list of replica nodes IP-addresses (endpoints) that are responsible for storing the needed data via matching the token number of the data item with the token number of the nodes. The consistency option that is associated with each request specifies the number of replica nodes to be contacted to succeed the operation. The coordinator node first ensures whether there is a sufficient number of replica nodes reachable to succeed the request. If a sufficient number of replica nodes are not available, the coordinator node fails the request immediately. During write operation, the operation will be forwarded to all the replica nodes that are alive and wait for the write acknowledgement from a sufficient number of nodes before issuing the success message to the client. If the write acknowledgement is not received within a certain time-out period, the coordinator node fails the operation.

Read operations follow a similar pattern, except that the request will not necessarily be forwarded to all the replica nodes. The coordinator node sends the read request to the closest replica node by sorting the replica nodes via proximity. The Snitches [Doc15b] in Cassandra helps to sort the replica nodes via proximity. Then in order to ensure the consistency option, the coordinator node requests the *md5* hash of the read response from the next closest replica nodes. The coordinator node uses the md5 of the read responses in order to ensure whether all the replica nodes hold the same data. If the md5 of the read responses do not match, the coordinator will send a read request to all the replica nodes in a second trip and respond to the client with the data that is higher in the timestamp. The

system follows a similar procedure during read repair, which is periodically initiated by the system based on the *read repair chance* configured per column-family as a part of eventual consistency guarantees. Requesting only md5 of the read response instead of requesting the full response is an optimistic approach chosen by Cassandra for read optimization. However, in case of md5 mismatch, requesting the actual value in the second trip in order to respond the client with the data that is higher in the timestamp is expensive in terms of read latency.

5.1.2 LibRe implementation inside Cassandra

LibRe protocol is implemented inside Cassandra release version 2.0.0. In the native workflow, while retrieving the set of endpoints addresses via matching the token number of the data against the token number of the nodes, the first endpoint address before sorting it based on proximity becomes the LibRe node for that data. As data are equally distributed among each node in the cluster, each node maintains a registry for a set of data items. Hence, every node in the cluster plays two roles: replica node and LibRe node for a set of data items. A separate thread pool for the LibRe messaging service is designed to handle the LibRe messages effectively. On system initialization, all LibRe registries are empty until a write request is executed, which will trigger the protocol and start filling up the registries. CaLibRe can be configured to work either by passing a list of data items in a configuration file, or by specifying directly the name of the column family/ table(s) to monitor. In the LibRe implementation, the write operations follow the native workflow of Cassandra, except that the replica nodes send a 'LibRe-Announcement' message to update the LibRe registry at the end of the successful write (if the data has to be managed by LibRe).

The 'LibRe-Announcement' message contains metadata about the write operation such as the *data-key* associated with the write, IP-address of the replica node, and a *version-id*. In the initial implementation of LibRe inside Cassandra, the hash of the modifying value is used as the version-id, although the hash value is not monotonically increasing and can not be compared directly to an existing version-id. When the LibRe node receives a 'LibRe-Announcement' message, the LibRe Announcement manager will take control

of it and handle the message. The messaging service will un-marshal the needed data and transfer it to the LibRe announcement manager to update the LibRe registry. The registry is updated matching the version-id of the operation with the version-id of the data already registered in the LibRe registry as described in Algorithm 3 of Chapter 3, except the comparison of the two version identifiers. In the prototype implementation of the protocol, assuming there is no delay in the update messages, the version-id that does not match with the version-id of the registry is considered to be the new version of the data. However, in the future implementation, the hash of the modifying value will be replaced by the timestamp of the operation, which increases monotonically and facilitates the comparison.

During read operation, if the data-key (Cassandra Row-key) is set to be managed by LibRe protocol, instead of following the default path in querying the request, the node will send a 'LibRe-Availability' message to the LibRe node. The LibRe-Availability message is prepared as the normal 'Read Message' for the needed data with the data-key on top of the message. When the LibRe node receives a 'LibRe-Availability' message, the LibRe availability manager finds an appropriate replica node to query the data. The data-key added on top of the message helps to find the target node without un-marshaling the whole message. After finding a target node to query the data, the read message will be sent to the appropriate node extracting the 'Read Message' from the 'LibRe-Availability' message. Handling the 'LibRe-Availability' message in this way minimizes the time to un-marshal and marshal the message and speedup the read response process.

5.2 YCSB for evaluating data consistency

Yahoo ![1] developed a benchmark tool called *Yahoo ! Cloud Serving Benchmark (YCSB)* in order to benchmark the performance of Yahoo's PNUTS system [CRS+08] over other cloud data serving systems. The reason behind the development of the YCSB in addition to existing TPC like benchmark tool is due to the generation gap between the traditional database systems and modern storage systems in terms of application workload and database functionalities. The traditional database systems are based on ACID properties [Ram03],

1. `http://labs.yahoo.com`

whereas modern storage systems replies on BASE (Basic Availability, Soft State, Eventual Consistency) paradigm [Pri08]. Although YCSB provides different means to evaluate modern storage systems in terms of Latency and throughput, it does not provide any means to evaluate consistency options. In this section, we present an extension to the YCSB codebase that includes a new feature to evaluate the number of stale reads exhibited by different consistency options along with the default evaluation metrics provided by YCSB. And with the extended YCSB, we compare the Consistency-Latency tradeoffs of CaLibRe (LibRe consistency option inside Cassandra) against different consistency options offered by Cassandra.

5.2.1 YCSB

The main goal of YCSB is to evaluate the performance of different types of modern storage systems under similar workload patterns. As the characteristics of the modern storage systems vary, the querying interfaces also differ for each storage system. Hence, using a single benchmark tool for all kinds of storage system could be challenging. In order to overcome this challenge, YCSB consists of two layers: the core YCSB layer and the database interface layer. YCSB users can define the test suite with the numbers of records, online users, queries per second and workload pattern, which are taken care of by the core YCSB layer. The second layer: Database Interface layer gets the input from the core YCSB to query the appropriate database and logs the query performance back to the core YCSB. The status evaluation module of core YCSB in return evaluates the performance of the test suite using the logs produced by the database interface layer. Since each database has different interface, a Database Interface layer is developed exclusively to query a specific storage system. The advantage of YCSB is the availability of various database clients and the easy extensibility of its code base.

The test metrics of YCSB are database-independent. Each test suite takes input on multiple parameters such as: IP addresses of the database instances, number of records to be stored, the size of the records, the number of users, the number of queries per second and the Workload pattern. The results of YCSB test suite contains Read and Write latencies of the system under various aspects such as Minimum, Maximum, Average, 95th percentile,

99th percentile latency.

YCSB analyzes all the needed metrics and tradeoffs of modern storage systems in terms of scalability and throughput. However, YCSB fails to provide a means to evaluate one of the indispensable tradeoff of modern storage systems: Consistency-Latency tradeoff. One issue in extending the YCSB framework to add a new evaluation metric along with the offered metrics is that, after each query, the database interface layer returns the needed metrics to the status evaluation module instantaneously. After all the operations are completed, the status evaluation module computes the needed metrics and prints the evaluation of the test case to an output file. So a pact between the two layers should be established safely without affecting the existing computations in order to compute a new metric.

As stated by Daniel Abadi in [Aba12; ABA10], in the absence of network partition, it is obvious to achieve both Consistency and Availability at the same time. The compromise arises only in case of network partitions. Network partitions could be experienced not only via partitions in the network, delays and failures of one or more nodes in the cluster also mimic the Network Partitions [CST+10]. As the probability of seeing the number of stale reads in presence of Network Partition is high, we choose to evaluate the performance of the storage system under this condition. We simulated network partition in the system via partial update propagation in order to evaluate the system performance under this situation. These information are described more in Section 5.3.1

5.2.2 Extension to YCSB

Since YCSB by default evaluates performance metrics of modern storage systems under various workload patterns, it is chosen to be extended for reporting the number of stale reads. In order to count the number of stale reads in real time, the benchmark tool must be aware of the most recent version of each data. Thus, during read operation, the version number of the read data can be compared to ensure whether the version number is the most recent version or not. If the version number of the read data does not match with the most recent version number, then it is considered to be a stale read. As mentioned in Section 5.2.1, the database interface layer of YCSB applies the write and read operations

on the target database and returns the needed metrics to the core YCSB layer for the status evaluation.

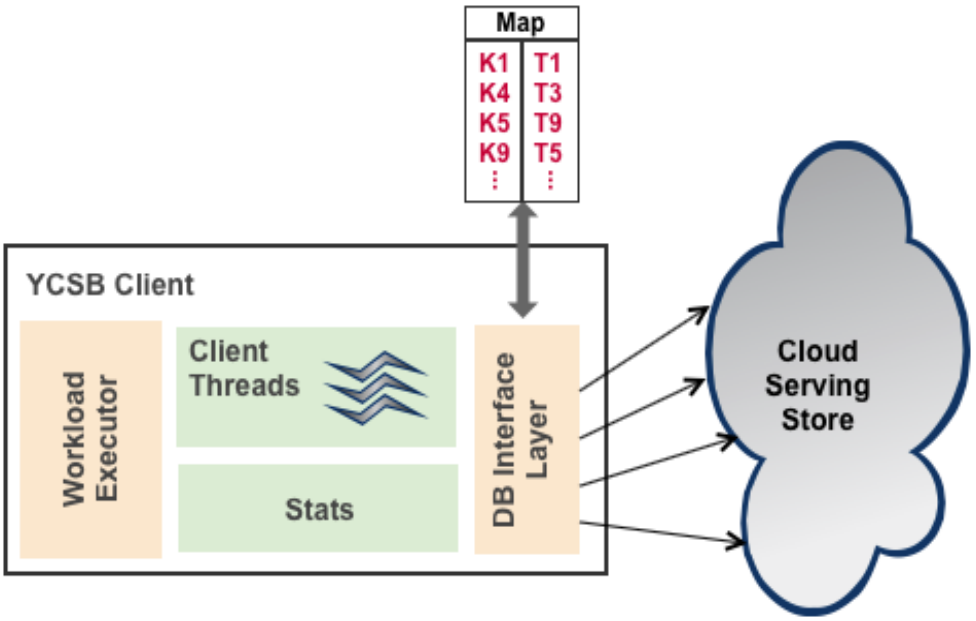


FIGURE 5.2 – Extended YCSB

In order to track the stale reads, a Hash-Map data structure is introduced in the database interface layer as shown in the figure 5.2 in order to account for recent version of a data item. During write (insert, update, delete are technically same) operations, the *data-key identifier* becomes the key of the Hash-Map with the timestamp of the write operation as its value. During read operation, the timestamp of the read value will be compared with the one recorded for the corresponding identifier in the Hash-Map. If both the timestamps are same, the read is considered to be consistent, else the stale read count will be incremented by one. In case of retrieving more than one column in a read operation, each stale column increments the stale read count. After each operation, along with the default evaluation metrics of YCSB, a stale read count will be returned to the status evaluation module. By this way, the status evaluation module prints the number of stale reads encountered during the test suite in the output file along with its default statistics. As the Hash-Map accounting the recent version of each data item has to be managed by a single machine, this approach only works in a centralized test setting.

5.3 CaLibRe performance evaluation using YCSB

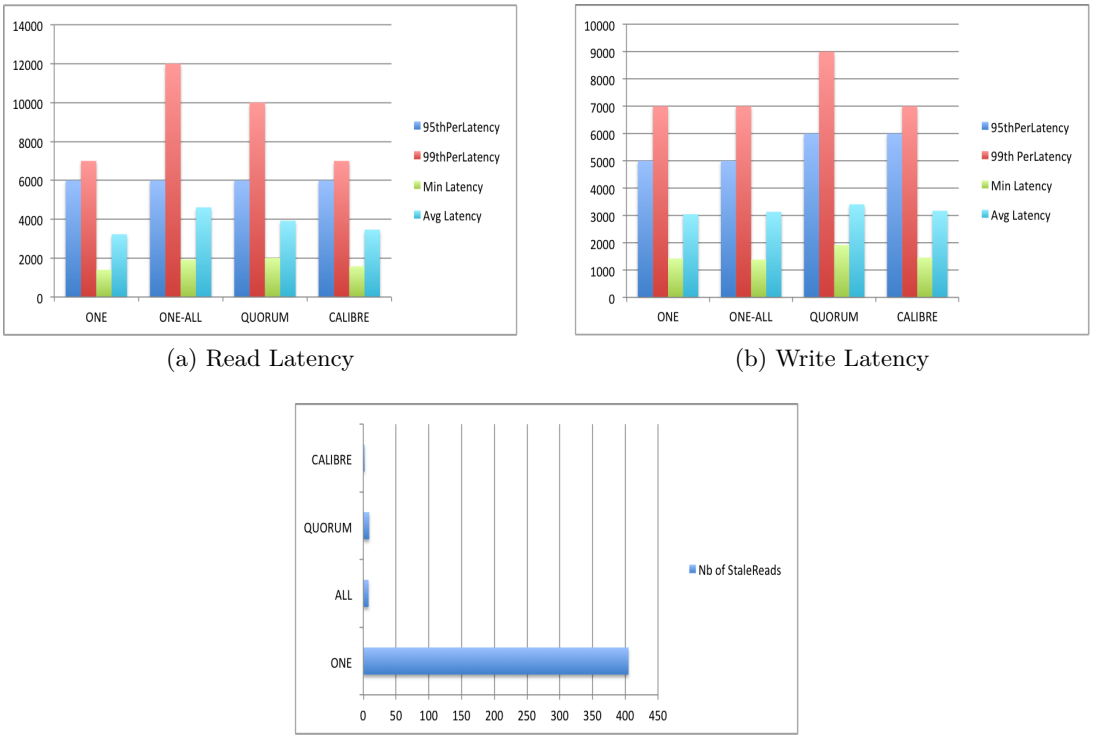
In this section, we will show the performance results of CaLibRe against Cassandra's native consistency options under same setup.

5.3.1 Test Setup

The experiment was conducted on a cluster of 19 Cassandra and CaLibRe instances that includes 4 medium, 4 small and 11 micro instances of Amazon EC2 cloud services and 1 large instance for the YCSB test suite [CST+10]. All instances were running Ubuntu Server 14.04 LTS - 64 bit. The workload pattern used for the test suite was the "Update-Heavy" workload (*workload-a*), with a record count of 100000, operation count of 100000, thread count of 10 and the Replication-Factor as 3. YCSB by default stores 10 columns per RowKey. We used RowKey as the data-key, for which an entry will be managed in the LibRe Registry. Since a RowKey can represent more that one column, using RowKey as the data-key could lead to a situation where, if one or few columns of a RowKey is updated on a replica node r_n, the registry will assume that r_n contains the recent version for all the columns of the RowKey. In order to avoid this situation, we configured YCSB to update all the columns during each update. The test case evaluates the performance and consistency of the 19 Cassandra instances with different consistency options (*one*, *quorum* and *all*) against 19 CaLibRe instances with a consistency option *one*. Performance is evaluated by measuring read and write latencies and consistency is evaluated for each level by counting the number of stale reads. In order to simulate a significant number of stale reads, a partial update propagation mechanism was injected into the Cassandra and CaLibRe clusters to account for the system performance under this scenario [KLCS14]. Hence, during update operations, instead of propagating the update to all 3 replicas, the update will be propagated to only 2 of the replicas.

5.3.2 Test Evaluation

Figures 5.3a, 5.3b and 5.3c respectively show the evaluation of Read Latency, Write Latency and the number of Stale Reads of Cassandra with different consistency options



(a) Read Latency

(b) Write Latency

(c) Stale Reads

FIGURE 5.3 – CaLibRe Performance Evaluation under Partial Update Propagation

against CaLibRe (Cassandra with LibRe) protocol. In figure 5.3a, the entity *one* represents the read and write operations with consistency option ONE. The read and write operations with consistency option QUORUM is indicated by the entity *quorum*. The entity *one-all* represents the operations with write consistency option ONE and read consistency option ALL. The entity *calibre* represents our implementation of the LibRe protocol developed inside Cassandra. Due to the injection of the partial update propagation, ROWA (Read One, Write All) principle could not be tested, as writes would always fail.

The read latency graph in figure 5.3a, shows that the 95th Percentile Latency of *calibre* is similar to the other consistency options of Cassandra. The 99th Percentile Latency of *calibre* and Cassandra with consistency level *one* remains same and better than the other options *one-all* and *quorum*. The minimum and average latencies of *calibre* are slightly higher when compared to Cassandra with consistency level *one* but better than the consistency options *quorum* and *one-all*. This is due to the fact that LibRe protocol imposes an additional call to the registry for all requests.

The write latency graph in figure 5.3b shows that the 95th percentile write latency of *calibre* is the same as the 95th percentile latency of *quorum*, and *calibre* is faster in other metrics: 99th Percentile, Minimum and Average latencies of *quorum*. However, while comparing to the entities *one* and *one-all*, some of the write latency metrics of *calibre* are slightly higher (but are not significant). This is due to the fact that both in *one* and *one-all*, writes need only one acknowledgement from a replica node. In CaLibRe also writes need only one acknowledgement, but there is an extra messaging (advertisement message) in the background.

Graph 5.3c shows the number of stale reads for each level of consistency. Cassandra with consistency level *one* shows the highest number of stale reads. There were a few stale reads in the other consistency options, but these numbers are negligible when compared to the total number of requests. From these results, it is possible to conclude that CaLibRe offers a level of consistency similar to the one provided by the *quorum* and *one-all* levels with better latency.

5.4 Conclusion

The proof-of-concept of LibRe protocol was implemented inside Cassandra storage system as an additional module to Cassandra code base. This so-called 'CaLibRe' implementation offers LibRe protocol as an additional consistency option for Cassandra storage system. The implementation let the user/application to switch between LibRe protocol and the different consistency options that are natively offered by the Cassandra storage system. The enhanced version of YCSB benchmark tool that provides an additional metric regarding database consistency was used to benchmark the CaLibRe performance. The experiment was conducted separately on a cluster of 19 nodes CaLibRe and Cassandra instances. Since encountering number of stale reads in a small test setup is minimum, partial update propagation is injected into the test setup to account for the system performance under this scenario. The performance metrics considered for evaluation are the number of stale reads encountered versus Minimum, Average, 95th Percentile and 99th Percentile of read and write latencies. Using the metrics, the consistency-latency tradeoff of CaLibRe was evaluated against Cassandra's native consistency options *one*, *all* and

quorum. The performance results prove that CaLibRe provides lower request latency compared to the strong consistency levels offered by Cassandra, combined to a similar number of stale reads. Hence, we can safely conclude that the LibRe protocol gives a new tradeoff between consistency, latency and availability. However, the performance results were not tested under nodes joining or leaving the clusters. During such events, LibRe protocol would experience temporary inconsistency, which has to be studied in future works. Another perspective to this work is to study the influence of the nature of the version-id (timestamp, version vector, vector clocks, ...). Although additional works are required to optimize the performance of the CaLibRe implementation, it is enough to conclude from the results obtained via simulation and benchmark that LibRe offers better tradeoff between data consistency and performance of the system. With this conclusion, the next chapter will describe about the application of LibRe protocol for adapting the consistency needs of replicated data.

Chapter 6

Overriding Application-Defined Consistency Options during Run-Time

As discussed in the previous chapters, one of the major limitations of the modern database systems is the inevitable tradeoff between Consistency, Availability and request Latency. In order to overcome this limitation, most of the modern database systems are equipped with a new feature called 'Adaptive Consistency'. By using adaptive consistency, these storage systems offer different consistency options and let the user switch the consistency guarantees of the system per query/data basis. Using this feature, an application developer can fine tune the consistency and performance of the system by sorting data that need strong consistency and the data that can loose the consistency guarantees.

However, one of the main challenges of the adaptive consistency feature is to decide on the consistency level of a data in advance as the user and/or system behavior could change over time. In order to overcome this limitation, there are enough works in the literature that try to identify the needed consistency level of a query/data during request execution time, instead of pre-configuring it. These works can be of two types. The first approach is to adapt the consistency-level of the whole system depending on the various environmental factors. In the second approach, the consistency adaptation is fine tuned to per query/data basis instead of adapting the consistency-level of the whole system. In order to adapt the consistency-level of the system per query/data basis during run-time, most of the existing works studied the use of a centralized service that helps to query the

requests with a needed consistency option.

The limitation of this approach is the intervention of the centralized service each time while querying the database could become the performance bottleneck of the storage system. In this chapter, we propose a system model that will eliminate the intervention of the centralized service during query time, but instead, move the participation of the service to the background. In this chapter, we also show the application of the LibRe protocol in the context of adaptive consistency. This chapter is organized as follows. In the next section, we give a brief summary about a few use cases that need to ensure stronger consistency guarantee for certain application queries for a temporary period of time. Section 6.2 gives more insight about the adaptive consistency feature that is followed in most of the modern database systems. Section 6.3 describes the proposed model to override the consistency options of the application queries at the data store level during query execution time. The experimental use case used for evaluating proposed model along with the experimental evaluation are discussed in Sections 6.4 and 6.5 respectively. The conclusion and future works of our proposed model are discussed at the end of this chapter.

6.1 Sample Use Cases

As discussed in Chapter 2, adaptive consistency depends on identifying the criticality of each query or data item in advance and specifying the needed consistency option along with the query. Kraska et al. in [KHAK09] showed that the consistency needs of a data can be classified into three categories such as Category A, B and C. The data under category A are always sensitive and hence these data will always demand strong consistency. The data under Category C always remain insensitive and thus the consistency guarantees of the data under this category can be relaxed all the time. Whereas, the consistency needs of data under Category B have to be decided during run-time depending on various environmental factors. If one or more factors influence the consistency needs of a data item and demands to augment its consistency need for a short period of time, we mention it as critical time frame of the use case/data item. In this section, we discuss different use cases and a short description about their critical time frames.

6.1.1 Inventory Control Systems

An inventory control system keeps track of the availability of a list of resources depending on the demand and supply chain. It is widely used in e-commerce websites to keep track of availability of each product and online reservation systems such as hotel and flight reservation. These systems reach a critical time frame when the availability of a resource goes to a critical number (say., less than 5). When resources are abundant, it is safe to process the queries related to the particular resource with a weaker consistency option offering reduced latency and high availability to the users. However, if the availability of a particular product goes beyond a critical number, we need to enforce a stronger consistency option for the queries regarding that particular resource. But it is still safe to process the queries related to other resources with a weaker consistency option. Hence, in these types of systems, the system has to adapt the consistency guarantee of the resources when the availability of the resources goes beyond a critical number and until the resources gather enough availability.

6.1.2 Auction Systems

Auctions are common in e-commerce websites to sell products. In auction systems, people who wants to sell a product can bid the product publicly (eg., via internet) quoting an initial price for the product and a deadline. People who are interested in buying the product can bid a higher price than the recent bid value. Finally, at the end of the deadline, the person who bid the highest price will be considered as the winner, and the product will be sold to him/her for the price he/she bid. In this type of systems, it is normal that for the first couple of hours/days till the auction sees enough popularity, there would not be much bidding for the product. However, during the last minutes before the auction ends, it is more likely that there will be high competition in bidding the product in order to win the auction. In these systems, if a user can not see the latest bid value of a product before the critical time (say., few minutes before the deadline), it is absolutely fine since other users will still have enough time to bid a higher price (if he/she wants to). But if all users cannot see the latest value during the last minutes of the auction, the user experience will be affected. Hence, the system has to ensure strong consistency on the bid values during

the last minutes before a particular auction ends, whereas a relaxed consistency is enough for the rest of the time.

6.1.3 Bike Sharing Systems

Bike sharing systems are finding enough popularity in modern cities in order to reduce pollution and traffic. The system is composed by a set of stations containing bikes that are deployed around the city. People who subscribed to the service are usually helped by a mobile application in order to find the nearest station and the number of bikes available in the stations. However, due to the limited connectivity of mobile devices and to ensure faster response to the users, weaker consistency guarantee regarding availability of bikes in the stations is preferred. Due to weaker consistency guarantees, it may happen that the number of available bikes or free slots showing up in mobile applications for a requested station is not up to date. Although, the system provides high availability and faster responses to the users, sometimes the misguidance of the app can cause inconvenience to the users. The inconsistent numbers shown by the app may lead the users to waste their time and effort to move to a specific bike station containing no available bikes/ slots. Moreover, the users can be charged an additional fee if the bikes are not parked on time. Depending on the geographical location of the stations and time, only a fraction of stations will get high clients interest with high frequency of bikes coming and going when comparing to the other stations [CF14]. Hence, the system can ensure stronger consistency only for the stations that has unbalanced number of bikes coming and going, whereas weaker consistency guarantee is enough for the rest of the stations.

6.1.4 Emergency Situation

Emergency environments can apply to any type of use cases that need immediate, unexpected attention that demands to override the consistency option of a data item to a stronger consistency option for a short period of time. For example, in case of a navigation system that includes real time traffic information about the routes, weaker consistency guarantees are normally enough to provide additional traffic information. However, in case of redirection of route due to accident or administrative reasons, the real time traffic

information about the specific route needs immediate attention (with strong consistency) to all the users who use the particular route information. Once the traffic/ route is cleared, the consistency guarantees of the information related to the specific route can switch back to the weaker consistency guarantee.

6.2 Learning Consistency Needs

Modern storage systems that facilitate adaptive consistency use quorum-based replication for replica management. As described in Chapter 2, consistency of a data item is ensured depending on the number of nodes contacted during write and read operations. The systems that rely on eventual consistency depend on contacting a minimal number of nodes in order to reduce request latency and ensure availability of the system. The minimum number of nodes to be contacted to proceed with a read or write operation is 1. The maximum number of nodes that need to be contacted to ensure strong consistency is equal to the number of replica nodes configured for the particular data item. Usually, the application developers decide this number in advance and query the needed data items by passing appropriate settings to the database.

As most of these systems rely on peer-to-peer architecture for scalability reasons, a client can connect to any node in the cluster and issue the query with the specified consistency option. In this case, the node to which the client connected to becomes the coordinator node for the particular request and is responsible for querying the request from the needed number of replica nodes. Figure 6.1 describes this principle.

Although these systems facilitate adaptive consistency, deciding the consistency option for a particular data/ query in advance during application development time becomes tedious for the application developers. For this reason, most of the adaptive consistency systems decide the needed consistency option for the queries dynamically during query execution time with the help of a separate service deployed on top of the eventually consistent data stores [CIAP12; YV00; LLJ07]. These services monitor the behavior of the application and the storage system and detect the needed consistency option of the requests. Some of the factors that could influence the consistency need of a request/data

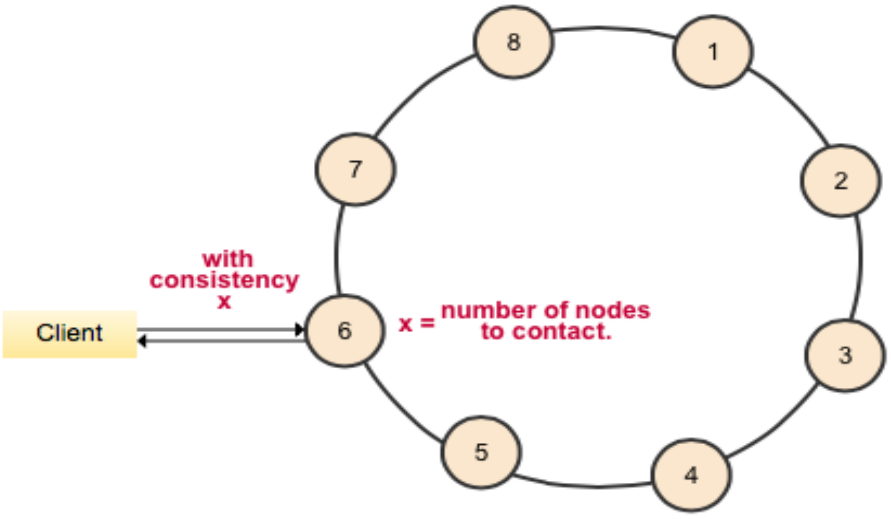


FIGURE 6.1 – Tunable Consistency model

include request latency, probability of inconsistency, application patterns, performance throughput etc. Routing the queries through one of these services helps to choose an appropriate consistency option for the requests during run-time rather than application development time. However, the problem of using a separate service for deciding the needed consistency option during run-time demands each request to pass through this centralized service. In this case, the system performance will be narrowed down to the performance of this centralized service.

6.3 Proposed Model

In order to overcome the above limitation, we propose to enable consistency adaptability inside the storage system itself instead of relying on a third-party service that is external to the storage system. The main goal of our model is to provide a mean to augment the consistency guarantee of a data item from weaker to stronger consistency level. Hence, the application developers can choose strong consistency only for the sensible data items. For the remaining data items, default eventual consistency (consistency option: ONE) can be provided during application development time, and could be overriden during run-time when needed. In order to override the weaker consistency level of a data to a stronger level during run-time, we rely on external inputs. In our model, the external

input includes information about a list of data identifiers (data keys) along with a time period (with start and end time) and a needed consistency option. For example, as studied by Chihoub et al. in [CPAB13], the input can be computed based on the access pattern of the application using machine learning techniques (say., Clustering). Based on this input, the system will apply the specified level of consistency to each given data item for the configured time period.

The input can be provided either by an external service or by a database administrator via connecting to one of the nodes in the cluster and the node will then broadcast the information to all the coordinator nodes in the cluster. When a node receives this information, the data items that have similar time periods and consistency option will be grouped and packed into a bloom filter [BMM02] marking it with appropriate timestamp and consistency option. Although there could be multiple bloom filters depending on the timing information, only the bloom filters whose start and end times that correspond to the current wall time will be in active mode.

When the start time of a particular bloom filter is equal to or greater than the current wall time, the bloom filter will be in active mode. When the time period of a particular bloom filter reaches its end time, the bloom filter will be demoted to inactive mode and then eventually deleted. Two or more bloom filters that are similar in time period and consistency option have to be merged accordingly to form a single bloom filter. Note that, there could even be no active bloom filter at some point in time. The reason for choosing bloom filter for this action is that bloom filters are fast and space efficient data structure and the *false positive* chances of the bloom filter [BMM02] do not cause any impact to the use case. In our model, as we are trying to augment the weaker consistency guarantee of certain data items based on its existence in one of the active bloom filters, in case of *false positive*, the system will apply strong consistency to some data items for which weaker consistency is sufficient. This is absolutely fine in our case.

During a get or put operation, when a client connects to one of the nodes in the cluster and issues an operation, the particular node (coordinator node) will check locally whether the particular data item exists in one of the active bloom filters (if any). If the data item that needs to be queried exists in one of the active bloom filters, the incoming request

will be executed with the consistency option that the bloom-filter represents. If the data item does not exist in any of the active bloom filter, the request will be executed with the default consistency option that comes with the request.

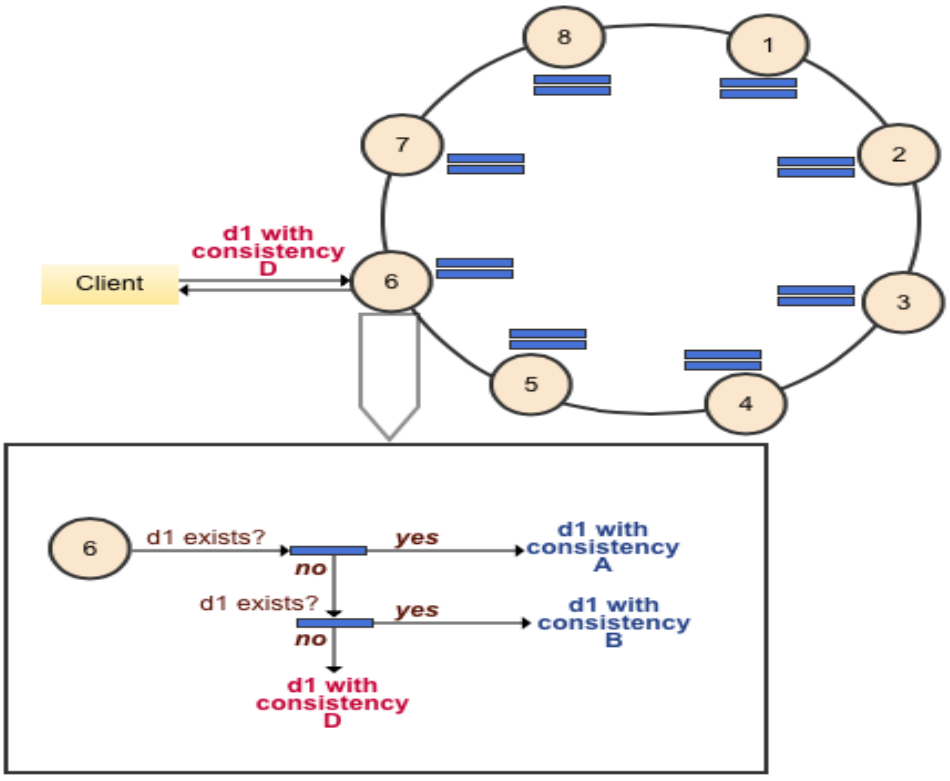


FIGURE 6.2 – Overriding Application-defined Consistency options

As in Figure 6.2, each node in the cluster contains some active bloom filters (2 in the example of the figure). When a client issues a query with a default consistency option (option D in the example), the coordinator checks whether the data item that needs to be queried exists in the first bloom filter. If yes, the particular request will be queried with a consistency option that is represented by the bloom filter (option A in the example). Else, the same process will be continued for the next set of active bloom filters. If the data item does not exist in any of the active bloom filters, the request will be queried with the default consistency option of the request (option D in the example).

6.4 Adaptive Consistency for Bike Sharing System

In this section, we describe adaptive consistency needs for the bike sharing system use case that is described in the Section 6.1.3. We discuss a prototype implementation of our proposed model inside Cassandra along with the experimental evaluation. The bike sharing system that we studied for our experimentation is the *Velib'* system [Vel]. Velib' is a bike sharing system deployed in the city of Paris. As described in Section 6.1.3, the system involves a set of stations with bikes, and users are assisted by a mobile application that sometimes provides an inconsistent number of bikes and free slots available in a particular station. Depending on the geographical location of the stations, few stations will get high clients interest with high frequency of bikes coming and going when comparing to the others. Hence, handling of bikes for highly demanded stations is a problem for the Velib' system. Several works have studied this case. In [CF14] Chabchoub et al. use k-means clustering with DTW distance and show that Velib' stations can be classified into three categories depending on the frequency of bikes added and taken to/from a station.

— Balanced: The stations belonging to this cluster have similar (balanced) frequency for the number of bikes added to the station and the number of bikes leaving from the station. In this case, the velib' system does not need to show special care regarding the bike regulations (rearranging the bikes), because there will be always enough number of bikes and free slots available in these stations.

— Under-Loaded: These stations have very few or no bikes left, which is due to the number of bikes taken from the station is higher than the number of bike added (parked) to it by the users. In this case, the Velib' system has to take care of making some availability of bikes for the users who wish to take some bikes from the station for the trip.

— Overloaded: These stations have few or no free slots left due to more number of bikes parked to the station than the number of bikes taken from it. In this case, the Velib' system has to make some room for the incoming bikes to get parked by shifting (rearranging) some of the bikes from these station to the stations that belong to the under-loaded cluster. The work of chabchoub et al [CF14] also gives an insight

of the time at which the system has higher frequency of bikes moving between the stations that lead to the formation of these under-loaded and overloaded clusters. This time is estimated between morning 6h00 and 9h00.

We have taken these inputs to experiment our proposed model to adapt the consistency needs of the velib' mobile application. The critical time frame for the Velib' system is considered to be between morning 6h00 and 9h00. Hence, during this time, the system has to offer strong consistency for the requests that are related to the stations that belong to overloaded or under-loaded clusters.

6.5 Experimental Evaluation

In this section, we will discuss about the prototype implementation of our proposed model inside Cassandra followed by experimentation test setup and results.

6.5.1 Prototype Implementation

In the prototype implementation, we consider only one bloom filter information for which the input is provided beforehand at the time of instantiating the storage instances. In a configuration file, we specify the file path that contains the list of station id's that belongs to the Overloaded and Under-loaded cluster and the time period during which the system turns into Overloaded and Under-loaded (morning morning 6h00 to 9h00). When Cassandra instances are instantiated, the station ids from an input file will be used to create Bloom Filter [BMM02] on each Cassandra instance.

During experimentation, while switching between stronger and weaker consistency models, we choose to use CaLibRe (the protocol described in Chapter 3) as the stronger consistency option and the consistency-level ONE as the weaker consistency option. According to the protocol, during write operations, each node checks whether the system clock time is equal to or greater than the start time and less than the end time of the bloom filter. If yes, the nodes check whether the data identifier (station id in our case) for which the write operation has been accomplished exists in the bloom filter. If both conditions are true, the system sends an advertisement message to the registry node (cf. 3.2.3) in

the background. Else no action will be taken. By this way, the registry information of the CaLibRe protocol will be built during the start time of the bloom filter and then stopped when the end time of the bloom filter is reached. During read operations, when a node receives a request, it checks whether the system time is between the time period specified in the configuration file and station id for which the information is demanded exists in the Bloom Filter. If both the conditions are true, then the data will be read following CaLibRe protocol as described in Section 3.2.4 of Chapter 3, else the default consistency level: Consistency-option ONE will be used for the read operation.

6.5.2 Test Setup

The data type used for the experimentation is the Register type that takes station-id as Key and the current number of available bikes as value. In Cassandra vocabulary, the station id is the row key and the available number of bikes in the station as the column value. We initially keep all the stations in the balanced state (cf. Section 6.4), so the available bikes in each station will be half of the total number of available slots in that station.

We used a cluster of 10 Cassandra nodes, which includes 2 medium, 2 small and 6 micro instances of Amazon EC2 [1]. For the test client, we used a separate medium instance. All instances were running on the platform Ubuntu Server 14.04 LTS - 64 bit. We evaluated the number of stale reads produced during the replay of Velib' utilization traces of 31 March 2013 that was provided by the concerned authorities of the Velib' system for research purposes. The trip data includes two types of information such as the time at which a bike is taken from a station and the time at which the bike is parked to a station along with the bike id and station id. The test client first loads the initial number of bikes and free slots available in each station in an in-memory Hash-Map. The test client then replays requests from the trip file that are sorted based on the timestamp. We have scaled the 24h trip data into 40 minutes in order to save time and experimentation cost.

The client processes two types of write operations and a read operation.

1. `https://aws.amazon.com/ec2/`

Write 1: When the client encounters a trip information about bike taken from a station, the client will read the available number of bikes in the specific station locally from its in-memory Hash-Map, decrements it by one and sends an update query (write operation) to the database and updates the local in-memory Hash-Map as well. The client records the write latency of the request in a log file once the acknowledgement for the write operation is received.

Write 2: In the same way, for the trip information about a bike parked to a station, the available number of bikes in the particular station will be incremented by one and the new number will be updated on both the local Hash-Map and the database at the same time. The client records the write latency in the log file once the write acknowledgement is received.

Read: After each query (both write 1 and write 2), in order to evaluate the number of stale reads, the client will randomly connect to one of the nodes in the cluster and issue a read for the number of available bikes in the station (using the station id) for which the write operation is just accomplished. If the number of available bikes in the station read from the database is consistent with the number in the local in-memory Hash-Map at the client side, the read will be considered as consistent, else the read will be counted as stale read. The client will record the read latency in the log file along with the information about whether the read is consistent or stale. It is also important to note that we have used partial update propagation inside the cluster in order simulate enough number of stale reads as described in Section 5.3.1 of Chapter 5 and account the system performance under this scenario. At the end of the test suite (after replaying 24h trip data), the log file will be processed to compute the total number of stale reads and other latency related metrics.

6.5.3 Test Evaluation

In figure 6.3, the x-axis denotes hours from 00h to 00h on the next day. The entity 1 represents the time from mid-night to 1 o'clock, entity 2 represents time from 1 o'clock to

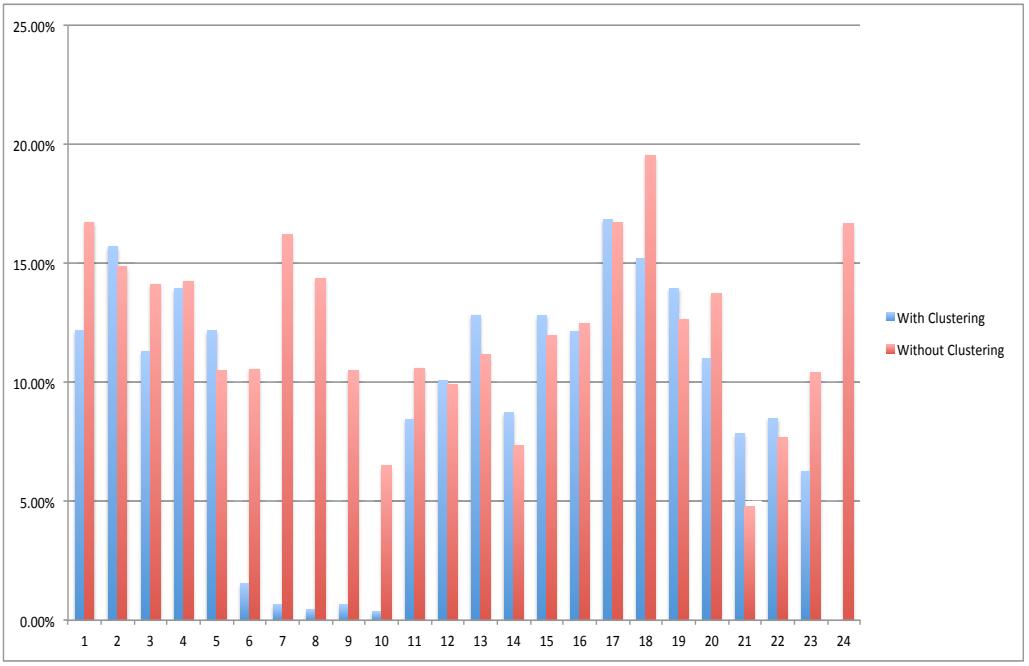


FIGURE 6.3 – Overriding Consistency Options of Velib system

2 o'clock and so on. The y-axis is the percentage of stale reads recorded during each hour. The red bar denotes the percentage of stale reads experienced by the Cassandra cluster that uses default consistency option ONE for all read and write operations: denoted in the chart as 'Without Clustering'. The blue bar denotes the percentage of stale reads experienced with our prototype implementation that uses strong consistency (LibRe) for the read requests that belongs to vulnerable station id's during the specified time period and the consistency option ONE for the rest of the cases: denoted in the chart as 'With Clustering'. Since the trip timings are scaled from 24h to 40 min and there could be a small time drift between test client and the server, the clustering time was extended to 5h - 10h instead of 6h - 9h. From the graph, we could see clearly that the number of stale reads recorded between 5h and 10h (entity 6,7,8,9,10) is almost negligible. And the stale read bars during rest of the time are similar to that of the red bars. This proves that the proposed system model helps to adapt the consistency-level of certain data item for a temporary period of time and relaxes the consistency-level for the rest of the time according to the provided information.

6.6 Conclusion

Although most of the eventually consistent systems facilitate adaptive consistency per query basis, deciding the consistency option of a query in advance during application development time remains challenging for the application developers. Choosing appropriate consistency options of the incoming queries dynamically during run-time taking into account different environment factors is a broad area of research. The existing works in the literature study various environmental factors based on which, the consistency guarantee of a query/data could be tuned during run-time. Most of these works suggest using a separate service on top of the eventually consistent systems to analyze the influential factors. By contacting the service during query time helps to choose an optimal consistency option for the incoming queries. The downside of this approach is that contacting the service each time while querying the database in order to choose an appropriate consistency option of the incoming queries would become a performance bottleneck of the system.

To the best of our knowledge, no work in the literature studied the possibility of enabling the modern database systems to adapt the consistency guarantees of the incoming queries or needed data on the fly at the storage system level instead of using an intermediate service. In this chapter, we proposed a system model that can enable the system to adapt the consistency options of the incoming queries with the help of an external input. This helps to override the consistency options of the incoming queries that are chosen during application development time with a needed consistency options that are predicted during run-time. The external input could be given by the database admin or by an external service directly to the system. By this way, the proposed model eliminates the intervention of a third-party service during query time, instead, pushes it to the background in order to augment the performance of the system.

The prototype of the proposed model was implemented inside Cassandra distributed database system and the working of the model was validated successfully for a bike sharing system use case. In the prototype implementation, we have focused a simple use case by providing the external input in advance via a configuration file. However, adapting the consistency options by receiving the external input from a database admin or by an

external service during run-time is considered for the future works.

Chapter 7

Application-defined Replacement Orderings for Ad Hoc Reconciliation

As most of the modern data stores choose to trade consistency in favor of request latency and availability, these systems let the replicas hold different values for the same key, accepting temporary inconsistency between the replicas. Identifying the most recent replica among different copies and propagate it to other replicas as fast as possible and achieving mutual consistency among all replicas demands effective data reconciliation. Data Reconciliation is the process of reconciling conflicting values of the replicas to reach a single value among all the copies [Mai08]. Conflict-free replicated data types (CRDTs) [SPBZ11b] introduced by Shapiro et al. are very useful for fast and efficient reconciliation of conflicting data values among the replicas based on the semantic notion of a specific data type. Since CRDTs are designed to work on the semantics of a specific data type, the notion of consistency and the reconciliation (merge) rules are fixed at system implementation time. The common CRDTs are registers, counters, sets, graphs, and object stores [SPBZ11a; SPBZ11b]. If none of these variants are applicable for the required semantics of the application, the developers usually fall back on one of the traditional semantic reconciliation techniques for detecting and resolving the conflicts [RD15b].

In this section, we introduce a register type named *priority register* that is parameterized by an application-defined replacement ordering, which will be useful for a broad class of use cases where existing CRDTs are not applicable. The following section (Section 7.1)

will provide a brief review of syntactic and semantic reconciliation techniques, providing motivation of our work in Section 7.2. In Section 7.3, we present the general concept and the special case of a priority register based on lexicographical orderings, following that, we provide the details about the proof-of-concept implementation of priority register inside Cassandra in Section 7.4. Conclusion and future work are discussed at the end of this chapter.

7.1 Syntactic and Semantic Reconciliation

Data Reconciliation involves verifying the correctness criteria to identify conflicts and resolve them according to the semantics of the external objects that use the data item [DGMS85]. Based on the conflict detection schemes, data reconciliation processes can be broadly classified into two types: Syntactic Reconciliation and Semantic Reconciliation [DGMS85; Mai08].

7.1.1 Syntactic Reconciliation

Syntactic Reconciliation techniques are usually based on the Serializability or Causality relations that are captured via logical clocks: Vector Clock [Fid88], Version Vector [PPR$^+$83] or sometimes via perfectly synchronised physical clocks. Syntactic Reconciliation techniques are fast, efficient and can be automatically handled by the system. However, these techniques are not a global option that could be used for all types of use cases. In addition, due to optimistic replication strategies, these techniques are prone to *syntactic conflicts* aka update conflicts that needs to be resolved via *Semantic Reconciliation*. Serializability relies on total order on updates, whereas Causality relies on partial order.

Serializability based syntactic reconciliation: Serializability-based syntactic reconciliation ensures strong consistency among the replica states. In distributed database systems, this can be achieved via various techniques such as primary copy replication [AD76], two-phase commit protocol [FGJ$^+$78], quorum-based replication systems [Vuk10] and consensus algorithms such as Paxos [CGR07] and Raft [OO15]. Zookeeper Atomic Broadcast

(ZAB) protocol [JRS11] is also one of the variants that uses serializability-based syntactic reconciliation.

These techniques demand tight coordination among all the system replicas during read and/ or write operations, adding bottleneck to the scalability and availability of the system. As stated by Eric Brewer in CAP conjecture [FGC+97; GL02a], in distributed systems, it is impossible to achieve Consistency, Availability and Partition Tolerance at the same time. Since it is inevitable for a distributed system to sacrifice partition tolerance [Hal10], the choice is even narrowed down to Consistency and Availability. Moreover, as described by Daniel Abadi in [Aba12] consistency works closely with request latency. For these reasons, most of the distributed systems choose to sacrifice consistency in favor of request latency, availability and scalability concerns. These systems compromise stale reads, conflicting writes and resolve the consistency issues at the background based on the causal ordering of the operations on the system.

Some systems, for example Cassandra [LM10], implement serializability-based syntactic reconciliation based on the timestamp of the client operations using synchronized physical clocks. These techniques do not affect the request latency, availability or scalability, but the consistency guarantees of such an approach are clearly problematic in case clock drifts [PBA+10].

Causality-based syntactic reconciliation: In distributed file systems, Locus [WPE+83] is one of the earliest system that used version vectors [PPR+83] for syntactic reconciliation of files. Each file in the system is associated with a vector representing the version number of the file in every replica nodes. When a replica node performs an update, it increases its version number by one. With this mechanism, the system lets the disconnected replicas handle updates to files independently and during replica synchronization, the version vector of a replica copy that is higher in some components (and equal in all others) will replace other replica copies. The decendents of Locus, such as Coda [KS93a] and Ficus [RHR+94] also used the same principle of syntactic reconciliation using version vectors.

Dynamo [DHJ+07a], which is one of the most influential distributed database systems implementing syntactic reconciliation by capturing the causality relation between

the updates, uses vector clocks [Fid88]. In Dynamo, each put operation contains three components such as a key, a value and a context (vector clock) representing the causal order [DHJ+07a] of the update. A vector clock is a version vector that is updated not only on local modification of the data but also on every synchronization message for this data. It enables the system to order the update operations that were performed on the data. Based on this causal order, it is possible to identify which version of a data object can replace another by taking the version with the highest counters in its vector clock.

Riak [Klo10; RD15a], which is a descendent of Dynamo in its latest version (from 2.0) uses dotted version vectors [PBA+10] instead of vector clocks for causality-based replacement. Clearly, the version vectors as well as vector clocks may not be comparable, which means that not all conflicts can be resolved in this way. Furthemore, the system typically only has a partial view of the real world causality, so causality-based syntactic reconciliation is necessarily incomplete.

7.1.2 Semantic Reconciliation

Semantic reconciliation techniques, which are often domain-specific, are normally needed in the following situations:

Syntactic conflicts: Update conflicts that occurred during syntactic reconciliation and that could not be resolved.

Other notions of consistency: Use case scenarios where conflict detection or reconciliation cannot rely on serializability or causality.

7.1.2.1 Syntactic Conflict Resolution

As serializability-based syntactic reconciliations offer tight consistency by default, these techniques are not subject to syntactic conflicts. Syntactic conflicts occur often in causality-based syntactic reconciliation techniques. When the version vector of a file or data object is higher in some of its components and lower in others, neither of the version vectors would dominate the other and this situation will be considered as an update conflict. In order to resolve these conflicts, the system may need some domain-specific information, which is normally provided by the application software or certain predefined rules, or sometimes

by the users themselves. Such domain-specific information enables the system to resolve conflicts according to the intended semantics of the data (hence, semantic reconciliation). Systems such as Ficus [RHR+94] and Coda [KS93a] provide application-specific resolvers (ASR) [KS93b; Pun94], which are programs that contain reconciliation rules for different conflict types that commonly happen in the system. If none of the reconciliation rules account for a detected conflict type, the system will notify the user by e-mail to perform a manual reconciliation. Apart from these, there are also some special tools and programs such as one described in [GMAB+83; How93] for reconciling conflicts in a file system.

Some systems like Dynamo maintain the conflicting replica values and return all the conflicting values to the user to perform resolution at the application side. The conflicts could then be resolved by exposing the conflicts directly to the user or could be resolved by the application automatically. If the user is directly involved in resolving the conflicts, there is clearly a negative impact on the user experience that could affect the popularity of the system. In the other case, if the application takes care of the conflict resolution without user intervention, additional effort during the application development is needed to handle all possible cases.

7.1.2.2 Other Notions of Consistency

Bayou [TTP+95] is one of the popular systems which is purely based on domain-specific conflict detection and resolution. The system provides an explicit knowledge about the operations that were performed during network partition and how to deal with it. Each write operation to the system includes a dependency check that specifies how to detect conflict for the particular operation and a function to resolve it. Also, read operations provide results along with information about whether the writes are committed or tentative.

Log Transformations [DGMS85] is a technique in which a system could let the partitioned replicas perform operations independently while reflecting the order of the operations in a log with respect to the timestamp. When the connection between the partitioned replicas is healed, a *rerun log* will be created by combining the operations from the logs of each partition according to the sequence of the timestamp. The rerun log would contain the needed operations to be rolled back in order to reach the required semantics of the

operations.

SqlIceCube [PSL03] semantic reconciliation system follows an approach similar to the log transformation method. Unlike the log transformation method, SqlIceCube orders operations based on the semantic relations between the operations. The reconciliation process in SqlIceCube includes two phases: the semantic inference phase and the reconciliation phase. During semantic inference phase, the preconditions and semantic relations between the operations will be extracted from the code of the operations. The reconciliation phase orders the operations in such a way that a maximum number of operations will succeed based on the inferred information from the first phase.

The Watchdog [BP88] extends the general semantics of the Unix file system with a user-defined semantics for individual files and also for a set of files inside a directory. Unlike ASR that are invoked whenever a conflict is detected, Watchdog allows to express the user-defined semantics and how to resolve a conflict [Pun94].

Operation Transformation [SE98] is one of the popular models in collaborative editing systems for maintaining consistency among replicated documents. This approach exploits the semantics about the consistency criteria that converge all the replicas to a same state when operations on a document (insertion and deletion of characters) are applied in a different order. The approach uses a transformation algorithm that adjusts (transforms) the position id of the characters in a document during insert and delete operations. The work of WOOT (WithOut Operation Transformation) [OUMI06] defines a consistency model for positioning the characters in a replicated document by representing each document character as so-called W-characters. The W-characters include additional information about positioning the characters with reference to its neighboring characters and to position the characters in case some unordered additional characters exist between the neighboring characters.

One of the significant improvements in replicated systems for defining a precise notion of consistency is the definition of typed objects (with a well-defined set of operations) that use type information to decide how to detect and resolve conflicts. The approach of Herlihy in [Her86] associates the operations on a file to a specific type such as a queue, a table, or a double buffer for efficient data replication in a quorum system. In particular, it includes

two parts: a serial specification that specifies the operations accepted by a specific data type and a behavioral specification that describes how to detect conflicts [DGMS85].

Replicated Abstract Data Types (RADTs) [RJKL11] define the semantics of the data types: arrays, hash tables, and linked lists for a replicated environment using properties such as operation commutativity and precedence transitivity. The data types defined by Shapiro et al. *Conflict-free replicated data types (CRDTs)* [SPBZ11a] use the properties commutativity, associativity, and idempotence of operations to ensure Strong Eventual Consistency (SEC) [SPBZ11b]. While the eventual consistency guarantees are prone to update conflicts, strong eventual consistency (SEC) guarantees avoids update conflicts.

The Bloom programming concept introduced in [ACHM11b] assists the programmer in the detection of possible conflicts and helps to avoid those conflicts already during the application development phase. Both CRDT and Bloom approaches intend for strong eventual consistency using the properties commutative, associativity, and idempotence. These properties are also referred to as ACID2.0 (Associative, Commutative, Idempotent and Distributed) [HC09].

In the ENCODERS project [WMJ+15], the application uses ordered replacement policies for discarding redundant data objects. This project is aimed at Mobile Ad-hoc Networks (MANETs) [LS], where nodes can be frequently disconnected and reconnected. Ensuring reconciliation in this context is challenging, and hence, the techniques developped in this approach are akin to semantic reconciliation. In the next section, we describe the solutions used in the ENCODERS project and how these served as a motivation for the work presented in this chapter.

7.2 Motivation

The motivation of this work comes from the *partially-ordered knowledge sharing model* for loosely-coupled distributed computing [SKT14]. Specifically, it has been used for data objects in the ENCODERS [1] content-based networking project [WMJ+15] and earlier for robust knowledge dissemination in delay-/disruption-tolerant networks [M-.08] and cyber-

1. Edge Networking with Content-Oriented Declarative Enhanced Routing and Storage

physical networks [KSKH10; SKT10; Net]. In the ENCODERS project, the content-based data object eviction technique aims to manage the system resources efficiently by locally discarding certain data objects based on metadata (i.e., attributes) associated with these objects. This technique is based on a replacement order for a class of data objects, under which, one data object can replace another one of the same class. In the simplest case that is currently implemented in the public version [enc], the replacement order on data objects is a lexicographical ordering defined by specifying the priority of relevance on a subset of the attributes.

In such a *lexicographical ordering*, two objects will be compared based on the value of the first component (of highest priority), in case of equality, the comparison will be continued to the second component (of next lower priority) and so on. Typically, this order will be defined so that the data object that has more information content for a particular application will replace the data objects with less content. In another interpretation, fresher data objects may replace obsolete objects if the creation timestamp is used as an attribute, again from the perspective of a particular application. The full replacement order that is defined by the application on all data objects is called *application-defined replacement order*. Note that while the lexicographical ordering is a total order on the relevant attribute vector, the full replacement ordering is a partial order, because no ordering relations exist between different classes. Furthermore, it should be noted that the induced ordering on the data objects can be partial even inside a class, because two different data objects may have the same relevant attributes. This simple approach supports the replacement of data objects according to the application-defined semantics, thereby enabling a form of semantic reconciliation instead of the serializability or causality-based replacement (syntactic reconciliation).

The ENCODERS project uses Haggle [Hag] as an underlying framework, in which data objects are stored and transmitted with metadata in form of attribute-value pairs [SSH$^+$07; SHCD06]. The lexicographical replacement order is defined on a subset of attribute-value pairs considering each attribute-value pair in this set as a component. In a typical ENCODERS application, mobile nodes generate situation awareness data that has to be efficiently disseminated to other mobile nodes, without assuming that the network is always connec-

ted (delay- and disruption-tolerant dissemination). Since more than one mobile node could generate the same information multiple times, whenever the system encounters two data objects of the same class, one of them may be evicted based on the application-defined replacement order so that only the maximal objects in this class will be maintained and further transmitted. Applications and use cases that involve application-defined replacement orders are studied in the articles [CMY+13; CMKS14; KGK+13; KKS+12].

Since ordered replacement is not the right policy for all data objects, ENCODERS is using additional utility-based techniques [enc14] in order to improve efficiency of caching and content dissemination. In this case a notion of utility is used to define an order that is used for prioritization instead of replacement. These extensions are however beyond the scope of this chapter.

As a specific motivation for this work, consider now an extension of ENCODERS where a DHT-based cloud database acts as a pool (warehouse) of data objects and intends to bridge two or more MANETs [LS] consisting of ENCODERS nodes. Clearly, such a database needs to be scalable to accommodate many data objects and requests from a large number of MANETs.

Moreover, querying such a database before each insert in order to implement the replacement order, i.e., checking the ordering between the incoming data object and all existing data objects is too expensive in both bandwidth and latency. Hence, in order to overcome the limitations and to facilitate fast and efficient semantic reconciliation we propose a new priority register which enables the data object replacement process to happen automatically on the database side without the intervention of the application (which in our motivating use case would be ENCODERS).

7.3 Priority Register

In general, we define a *priority register* as a multi-value register (MV-register) [SPBZ11a] that performs data reconciliation based on an application-defined partial replacement order on the values instead of the causal relations between the updates. Like an MV-register, which due to the partial order on the operations can contain multiple values that are maxi-

mal in the causal order, a priority register can contain a set of values, specifically several data objects that are maximal elements in the replacement order. In the special case of a lexicographical ordering with a single class of data objects, it can still contain multiple data objects if their relevant attributes are equal. Hence, for the purpose of our proof-of-concept implementation we further simplify this concept and focus on lexicographical orderings that use the timestamp as the last attribute to break the tie (if all other attributes are equal). In this case, the priority register will only contain a single data object as a last-writer-wins (LWW) register [2] [SPBZ11a].

In modern data stores, data objects are identified by a unique key that is mapped to a value. This value can be a record containing any number of fields. However, values of these fields are normally opaque to the system [SPBZ11a]. In the case of the priority register, we assume that some fields' values express domain-specific information that may be useful for semantic reconciliation, as formalized below.

Narrowing down the general definition for the purpose of this chapter, a *priority register P based on a lexicographical ordering* is defined as a tuple $P = (D, V, <, F, v, O)$, where: (1) D is an arbitrary set of *data objects*, (2) $(V, <)$ is a partially ordered set of *values*, (3) $F = \{f_0, f_1, ..., f_n\}$ is a set of *field identifiers*, (4) $v : F \times D \to V$ is a mapping from field identifiers and data objects to field values, (5) $O = \{o_0, o_1, ..., o_m\} \subseteq F$ is an ordered set, and (6) the set $\{v(o, d) \mid d \in D\}$ is totally ordered for each $o \in O$.

The *replacement order* $\prec_P$ is the order on D defined such that $d \prec_P d'$ iff $v(o_0, d)$, $v(o_1, d), ..., v(o_m, d) < v(o_0, d'), v(o_1, d'), ..., v(o_m, d')$ lexicographically. Given this definition, if $d \prec_P d'$ then we also say that the data object d' can replace d. As mentioned earlier, we make the simplifying assumption that all objects contain a timestamp as one of the fields in O to break ties.

An application defines the replacement order by simply declaring, for a given class of objects, the subset O of fields that must be considered in the order. Each field in this subset is associated to a priority number expressing its precedence in the order. This subset can be seen as the specification of a *lexicographical order* on the set of all data objects D.

2. `http://docs.basho.com/riak/latest/dev/using/conflict-resolution/`, June 2015

By carefully choosing the order O, the application designer can precisely define the behavior of the storage system during reconciliation. This order constitutes what we call an application-defined replacement order. This order is a parameter of the register definition, hence, whenever the system detects two data objects with the same key (i.e., belonging to the same class), the system compares the values of the fields of the two data objects in the declared order, and keeps only the version that is the highest in the order. Since the replacement order is set as the default order, the system should follow the same reconciliation process during read-repair and active anti-entropy phases [3] as well.

In some cases, it might happen that some of the values needed to compute the traditional lexicographical order are missing. In this case, the system considers the missing value to be the minimum value possible in this field (*conservative default assumption*). This allows the system to sort data objects even when the information is incomplete.

7.3.1 Comparison with CRDTs

CRDTs are defined by a join-semilattice which is associated with a binary *least upper bound (LUB)* function on the possible values of the data type [SPBZ11a; SPBZ11b]. Since the LUB is unique and satisfies the properties of associativity, commutativity, and idempotence, computing the LUB on replica states safely merges the values of multiple states without any conflict. Clearly, the value set of the data type has to be designed in a way to render the LUB appropriate for the semantics of the data type.

The basis of our priority register, which is the application-defined replacement order is more fundamental than the LUB that one can find in CRDTs. As a general model for distributed computing, partially-ordered knowledge sharing focusses on the replacement ordering rather than a particular reconciliation operator. The reconciliation operation does not have to be the LUB, and it does not have to be a binary operation. In fact, it does not even have to be a function, because it may take into account context beyond the scope of the data objects to be reconciled. Hence, there are differences between CRDTs and the priority register in the general case. These differences are mainly due to the fact that the

3. `https://docs.basho.com/riak/2.1.1/theory/concepts/aae/#Read-Repair-vs-Active-Anti-Entropy`, September 2015

priority register is based on a rather general domain-specific ordering, whereas CRDTs are based on a join-semilattice with a specific semantic reconciliation operation.

For arbitrary replacement orders, the LUB may not always exist, but for our lexicographical replacement orders, which are the focus of our prototype implementation, it is always well-defined for two elements in the same class, which is totally ordered by definition (after taking the timestamp into account). Hence, in this special case it would be mathematically equivalent to consider each class as a CRDT.

7.3.2 Sample Use Cases

This section describes a couple of use cases where the priority register can be applied. The first use case, *meeting room scheduler*, is taken from one of the Bayou system [TTP+95] use cases with a minor modification to the scenario. The second use case, *cyber-physical system* is inspired by joint work between SRI and ISEP on cyber-physical and content-based networks, specifically in the context of the ENCODERS architecture [WMJ+15].

7.3.2.1 Meeting Room Scheduler

The *meeting room scheduler* application is one of the simplest applications for understanding the use of the priority register. Let us consider a mobile application that helps to reserve a university meeting room via a reservation form, and assume the application is accessible in a disconnected mode. Since a particular room can only be reserved by one person for a specific time slot, certain domain-specific decisions might be needed for deciding who might get preference over another. One of the common decisions could be based on the position of the person in the university who submits the reservation request. The position of a university person who has access to the application could range from PhD students, professors, to administrative staff like department director and dean.

Based on the position, a rank for each individual can be assigned as Student: 1, Professor: 2, Administrative: 3. A lexicographical replacement order for this specific example could be defined by position and timestamp (in this sequence) for reservations that have the same room number and time slot (morning, afternoon, evening). Hence, by using this replacement order, when the system receives more than one request with the same room

number and time slot, the conflict will be resolved based on the position of the person submitting the request. And if the requesting persons have a similar position in the university, the preference will be given for the person who requested at an earlier time.

7.3.2.2 Cyber-Physical Systems

One of the typical use cases and the motivations for the design of the priority register are cyber-physical networks that demand domain-specific reconciliation. For example, consider a use case, in which the temperature of a large site or city is monitored via multiple sensors that are continuously moving and observed data is disseminated using an epidemic protocol [EGKM04]. Depending on their locations, some sensors might be more trustworthy than others. Therefore, the temperature data coming from these sensors is tagged with metadata such as the time at which the reading was taken (time), the precision of the reading (precision), and the sensor rank determining the trustworthiness of the sensor (sensor-rank).

If we want to maintain only one temperature reading for a given time, the replacement order can be defined lexicographically in the sequence: timestamp, precision, and sensor-rank for the readings that have the same location id and time. When the system receives sensor readings from different sensors at the same place and on the same day, it will only keep the temperature reading that is higher in the lexicographical order defined by timestamp, precision, and sensor-rank. In the case, when one or more fields in the replacement order (timestamp, precision, and sensor-rank) are absent, the missing field will be assigned a minimum value and the data will be reconciled accordingly.

7.4 Implementation of the Priority Register inside Cassandra

In this section we show how to leverage the design of Cassandra [LM10] to obtain an efficient implementation of priority registers with application-defined lexicographical replacement orders (that use the timestamp as the last component) directly inside the kernel of the distributed database.

7.4.1 Cassandra Read-Write Pattern

The design of Cassandra is derived from the Dynamo datastore [DHJ+07a], in which data objects are stored in a distributed hash table (DHT) [ZWXY13; Sit08]. Hashing the key of a data object and matching the resultant token number against the token number of the nodes in the DHT identifies the replica nodes to query for the data objects of interest. Cassandra, a column family data store, uses a unique key to identify the replica nodes holding related set of columns and a column name for identifying an individual column. Each column in Cassandra is a triplet, which contains a key, a value, and a timestamp. Cassandra's data model is write-optimized, that is generally writes are faster than reads. The operations write, update, and delete are handled technically as 'upsert writes' [Doc15a]. An update operation actually inserts a column with a new value and a delete operation inserts a so-called tombstone column with a recent timestamp. While reading or writing a column from/to MemTables and SSTables [LM10], when the system encounters two columns with the same name, a reconciliation process will take place between them. The reconciliation process first compares the timestamp of both columns and chooses the column with the latest timestamp, in other words it operates as a LWW-register. If both columns have the same timestamp, in order to break the tie, the column that has the highest value will get the precedence, however, this case practically never happens.

7.4.2 Priority Register Implementation

Cassandra's column families are physically sorted in storage in order to facilitate efficient column filtering and ordering [Kan14]. Hence, the structure of a column family is a map of key to a sorted map of column key to value, as shown in the following Java syntax: `Map<RowKey, SortedMap <ColumnKey, ColumnValue>>.`

Each entry in this map identifies a single row of the corresponding column family. Cassandra rows can be either wide or skinny [Cas14]. Skinny rows carry a value for a single column name, whereas, in wide rows, a column name and one or more column value(s) identify a value with a timestamp. The columns inside wide rows are also known as clustering columns or composite columns. The column name (column key) of the composite column is known as the *composite key* [Cas13]. The column name and value(s) associated

with the composite key are separated by a colon (:) and are considered as individual components. The maximum size of a composite key should be no more than 64Kb [Wik13b]. The *composite comparator* [Dat12] helps to compare the values inside the composite key, in order to facilitate efficient scanning and filtering.

The design of the priority register takes advantage of the composite comparator in order to implement the lexicographical ordering on the attributes of the data objects. Therefore, the priority register definition follows the declaration syntax of the 'wide row'. In Cassandra, by default, values are stored inside a column family as a LWW-Register. In our prototype implementation, we do not consider collection data types such as map, list, or set. In order to alter a column family to store the values as priority registers instead of LWW-Registers, two extra metadata properties have to be added to the column family definition while creating the column family as shown in the examples listing 7.3 and 7.5. The two metadata properties are *replacement_order_index* and *replacement_order_length*. Both *replacement_order_index* and *replacement_order_length* are of integer type, and they are used to encode the replacement ordering for the composite key as follows:

replacement_order_index specifies the index of the first component inside the composite key that is the replacement order

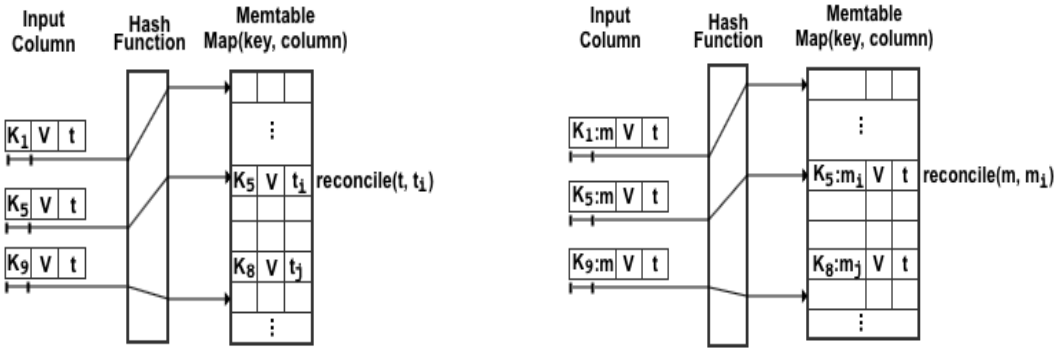
replacement_order_length specifies the total number of components including the first component that completes the lexicographical order.

In other words, the set of "replacement_order_length" components starting from the "replacement_order_index" component (included) inside the composite key forms the lexicographical replacement order. Since column family metadata properties are available on every Cassandra node, the ordering information is available everywhere in the system. The two metadata properties together with suitable modifications to the column family *SortedMap* class make sure that the attributes (replacement order) that are added inside the composite key are not considered part of the composite key during column reconciliation.

The default values of the *replacement_order_index* and *replacement_order_length* metadata of a column family definition are -1, which means the system behaves as normal, that is reconciliation is performed based on timestamp. If the values of *replacement_order_index* and *replacement_order_length* are set during column family creation,

the value of the two fields will be passed along with the comparator of the *SortedMap* class. The comparator of the *SortedMap* class masks the components inside the composite key that are indicated by *replacement_order_index* and *replacement_order_length* during get and put methods. Hence, technically the system can see only the components other than those specified by *replacement_order_index* and *replacement_order_length* as the composite key.

When encountering two columns with the same composite key, instead of reconciling the columns based on the default timestamp method, the columns will be reconciled based on the replacement ordering denoted by the sequence indicated by *replacement_order_index* and *replacement_order_length*.



(a) Standard Reconciliation (b) Priority Register Reconciliation

FIGURE 7.1 – Column reconciliation in Cassandra

As shown in Figure 7.1a, Cassandra input columns consist of three fields: key (composite key in case of wide rows), value, and timestamp, which are represented as K_n, V, and t, respectively. Columns are stored in Cassandra as key-value pairs with the name of the column as key and the entire column (containing K_n, V, and t) as value. If the MemTable already contains an entry for an input column name, reconciliation between the two columns will take place based on the timestamp associated with the columns.

In the case of our priority register implementation, shown in Figure 7.1b, each input column key (K_n) comes with the values of the metadata (m) associated with the lexicographical replacement order. While storing the columns, the values of the metadata are not considered part of the composite key. Hence, the part excluding the metadata (i.e.,

K_n) will be considered as composite key to store the values in the MemTable. If an entry already exists in the MemTable for a particular key, the columns will be reconciled based on the lexicographical replacement order defined on the associated metadata. The column whose metadata values higher in the order will then be stored in the MemTable. The system follows the same principle whenever two columns with a same name (i.e., K_n, not $K_n : m$) are encountered.

Although the embedded attribute values inside the composite key are masked by the comparator during column reconciliation, the attribute values are still visible for the normal purposes and at the client side as shown in the results of the examples in Section 7.4.3. When a tie between two columns cannot be broken via the defined replacement order, the system will use the default tie breaking mechanism, which is based on the timestamp.

Note that, the initial prototype implementation considers a total lexicographical order, in which only a single value needs to be stored. However, in a more general case, the priority register needs to store all data items that are equal in the lexicographical order. In that case, the default timestamp-based method for breaking the tie between two data objects that are equal in the order is not adequate. The future implementation of the priority register will address this case by storing multiple data objects that are equal in the lexicographical order, which also opens the door to additional reconciliation at the application level.

7.4.3 Prototype Sample Session

This section illustrates how the cyber-physical system use case described in Section 7.3.2.2 can be handled by our prototype implementation. We use the location id as the row key in Cassandra.

The following session (listing 7.1) shows an example with *skinny row* that contains single column name and a value:

```
create column family sensor_reading with comparator=UTF8Type
AND key_validation_class=UTF8Type;

set sensor_reading ['Loc1']['temperature']=32;
set sensor_reading ['Loc1']['time']=1050;
set sensor_reading ['Loc1']['precision']=double('8.5');
set sensor_reading ['Loc1']['sensor-rank']=double('5.2');
```

```
get sensor_reading['Loc1'];

=> (name=precision, value=8.5, timestamp=1437505096359000)
=> (name=sensor-rank, value=5.2, timestamp=1437505102951000)
=> (name=temperature, value=32, timestamp=1437505084943000)
=> (name=time, value=1050, timestamp=1437505090455000)
Returned 5 results.
```

Listing 7.1 – An example for skinny row

The session below (listing 7.2) shows a similar example with *wide row* (i.e., using a composite column), in which the value of the columns *timestamp*, *precision*, and *sensor-rank* are stored along with the *temperature* column and a *date* in order to sort the columns based on the date.

```
create column family sensor_reading with comparator =
'CompositeType(LongType,UTF8Type,LongType,DoubleType,DoubleType)'
AND key_validation_class=UTF8Type;

# The get and put method will be as follow:
# set sensor_reading['LocId']['date:temperature:time:precision:sensor-rank
    ']=reading-in-celcious;

set sensor_reading['Loc1']['110515:temperature:1050:8.5:5.2']=32;
set sensor_reading['Loc1']['110515:temperature:1050:8.5:5.3']=33;
set sensor_reading['Loc1']['110515:temperature:1051:8.6:5.2']=36;
set sensor_reading['Loc1']['110515:temperature:1051:8.5:5.2']=25;
set sensor_reading['Loc1']['110515:temperature:1050:8.6:5.2']=14;

get sensor_reading['Loc1'];

=> (name=110515:temperature:1050:8.5:5.2, value=32, timestamp
    =1437504279816000)
=> (name=110515:temperature:1050:8.5:5.3, value=33, timestamp
    =1437504285104000)
=> (name=110515:temperature:1050:8.6:5.2, value=14, timestamp
    =1437504298456000)
=> (name=110515:temperature:1051:8.5:5.2, value=25, timestamp
    =1437504293944000)
=> (name=110515:temperature:1051:8.6:5.2, value=36, timestamp
    =1437504289752000)
Returned 5 results.
```

Listing 7.2 – An example for wide row

The session in listing 7.3 shows how to specify a *priority register* with a lexicographical replacement order given by *time*, *precision*, and *sensor-rank* during column family creation. We also see the result of the same set and get methods as in the previous session. If we want to maintain only one temperature reading per day for a particular row

key, the value of the parameter *replacement_order_index* will be 3 and the value of the parameter *replacement_order_length* will be 3. The parameter *replacement_order_index* signifies that the index position of the replacement order for the composite columns starts from the third component inside the composite key, which is *time*. The parameter *replacement_order_length* signifies that the three components from the index position *time*, *precision*, *sensor-rank* are attributes of the sensor reading and the replacement order is the lexicographical order defined by this sequence. Hence, the remaining components inside the composite key, which are *date* and *temperature* becomes the actual name of the column (composite key). And the column reconciliation will be done based on the specified replacement order, when the system encounters two columns with the same name (*date:temperature*).

```
create column family sensor_reading with comparator =
'CompositeType(LongType,UTF8Type,LongType,DoubleType,DoubleType)'
AND key_validation_class=UTF8Type AND replacement_order_index=3
AND replacement_order_length=3;

set sensor_reading ['Loc1']['110515:temperature:1050:8.5:5.2']=32;
set sensor_reading ['Loc1']['110515:temperature:1050:8.5:5.3']=33;
set sensor_reading ['Loc1']['110515:temperature:1051:8.6:5.2']=36;
set sensor_reading ['Loc1']['110515:temperature:1051:8.5:5.2']=25;
set sensor_reading ['Loc1']['110515:temperature:1050:8.6:5.2']=14;

get sensor_reading ['Loc1'];

(name=110515:temperature:1051:8.6:5.2, value=36, timestamp=1437503817824000)
Returned 1 results.
```

Listing 7.3 – Priority Register example using thrift interface

The prototype implementation works with the CQL (Cassandra Query Language) as well. Specifying the same metadata properties *replacement_order_index* and *replacement_order_length* while creating the table is all that is needed to get the same result. But unlike thrift interface, which inserts one column at a time, in CQL more than one columns can be inserted in a single query. The first column name inside table's primary key will be considered as the row key and the remaining column names (so-called clustering key) inside the table's primary key (if any) along with a non-primary key column name forms a composite key. The session in listing 7.4 shows an example of how composite key(s) are formed from a table's primary key. We include a new colum *humidity* to our use case example for better understanding.

```
create table sensor_reading(
    loc_id text,
    date int,
    time int,
    precision float,
    sen_rank float,
    temperature int,
    humidity int,
primary key(loc_id, date, time, precision, sen_rank));

insert into sensor_reading(loc_id, date, time, precision, sen_rank,
   temperature, humidity) values ('Loc1', 110515, 1050, 8.5, 5.2, 32, 59);

# The above query will insert three composite columns as follows:

=> (name=110515:1050:8.5:5.2:, value=, timestamp=1437563035589000)
=> (name=110515:1050:8.5:5.2:humidity, value=0000003b, timestamp
    =1437563035589000)
=> (name=110515:1050:8.5:5.2:temperature, value=00000020, timestamp
    =1437563035589000)
Returned 3 results
```

Listing 7.4 – Mapping from CQL to Thrift interface

The first composite column with an empty value in the above session is the cql *row marker*. The implementation details about why cql inserts an additional row marker can be found at the [Cql12].

In order to define priority register with the same lexicographical replacement order *time*, *precision*, *sensor-rank* in CQL, the value of the parameter *replacement_order_index* will be 2, since the first column name inside the table's primary key is the row key and it should not be counted. Hence, the value 2 of the parameter *replacement_order_index* indicates the second component excluding the row key, which is *time*. And the value of the parameter *replacement_order_length* will be 3. The session in listing 7.5 shows a few examples of get and put methods in cql.

```
create table sensor_reading(
    loc_id text,
    date int,
    time int,
    precision float,
    sen_rank float,
    temperature int,
    humidity int,
primary key(loc_id, date, time, precision, sen_rank))
with replacement_order_index=2
AND replacement_order_length=3;

insert into sensor_reading(loc_id, date, time, precision, sen_rank,
```

```
    temperature, humidity) values ('Loc1', 110515, 1050, 8.5, 5.2, 32, 59);
insert into sensor_reading(loc_id, date, time, precision, sen_rank,
    temperature, humidity) values ('Loc1', 110515, 1050, 8.5, 5.3, 33, 60);
insert into sensor_reading(loc_id, date, time, precision, sen_rank,
    temperature, humidity) values ('Loc1', 110515, 1051, 8.6, 5.2, 36, 61);
insert into sensor_reading(loc_id, date, time, precision, sen_rank,
    temperature, humidity) values ('Loc1', 110515, 1051, 8.5, 5.2, 25, 70);
insert into sensor_reading(loc_id, date, time, precision, sen_rank,
    temperature, humidity) values ('Loc1', 110515, 1050, 8.6, 5.2, 14, 85);

select * from sensor_reading;

 loc_id | date   | time | precision | sen_rank | humidity | temperature
--------+--------+------+-----------+----------+----------+-------------
   Loc1 | 110515 | 1051 |       8.6 |      5.2 |       61 |          36


# Insertions with a missing non-primary key column

insert into sensor_reading(city, date, time, precision, sen_rank, humidity)
    values ('Loc1', 110515, 1051, 8.6, 5.1, 59);

select * from sensor_reading;

 loc_id | date   | time | precision | sen_rank | humidity | temperature
--------+--------+------+-----------+----------+----------+-------------
   Loc1 | 110515 | 1051 |       8.6 |      5.2 |       61 |          36

insert into sensor_reading(city, date, time, precision, sen_rank, humidity)
    values ('Loc1', 110515, 1051, 8.6, 5.2, 63);

select * from sensor_reading;

 loc_id | date   | time | precision | sen_rank | humidity | temperature
--------+--------+------+-----------+----------+----------+-------------
   Loc1 | 110515 | 1051 |       8.6 |      5.2 |       63 |          36

insert into sensor_reading(city, date, time, precision, sen_rank, humidity)
    values ('Loc1', 110515, 1051, 8.6, 5.3, 64);

select * from sensor_reading;

 loc_id | date   | time | precision | sen_rank | humidity | temperature
--------+--------+------+-----------+----------+----------+-------------
   Loc1 | 110515 | 1051 |       8.6 |      5.3 |       64 |        null
   Loc1 | 110515 | 1051 |       8.6 |      5.2 |     null |          36


insert into sensor_reading(city, date, time, precision, sen_rank, temperature
    ) values ('Loc1', 110515, 1051, 8.6, 5.3, 34);

select * from sensor_reading;

 loc_id | date   | time | precision | sen_rank | humidity | temperature
--------+--------+------+-----------+----------+----------+-------------
```

```
Loc1 | 110515 | 1051 |       8.6 |       5.3 |       64 |          34
```

Listing 7.5 – Priority Register example using CQL interface

7.5 Conclusion

Syntactic reconciliation techniques are fast and efficient for reconciling data objects in eventually consistent systems, but they cannot be applied to all use cases. Semantic reconciliation techniques are needed during update conflicts and for the use cases that demand domain-specific conflict resolutions. Although these techniques could be potentially applied to all types of use cases with domain-specific information, they are normally inefficient due to their complex resolution process. The idea of the priority register is to combine the benefits of both syntactic reconciliation and the semantic reconciliation techniques with the help of a very simple application-defined replacement specification.

In general, this replacement is defined by a partial order on the attributes of the data object, specifying the necessary condition under which one data object can be replaced by another of the same class. In this chapter, we have focussed on an important subclass of lexicographical replacement orders which covers many important use cases.

We have shown how, with relatively minor modifications in the Cassandra kernel, the basic LWW register can be transformed into the priority register by adding the domain-specific replacement specification. With this design, the client-side resolution rules can be moved directly to the server-side for efficient data reconciliation. By using the domain-specific information as a parameter of a data type, the problem of the concurrent updates (update conflicts) can also be eliminated for many applications.

The preliminary implementation of the priority register inside the Cassandra distributed storage system is intended as a proof of concept. The implementation is added as an extra module to the Cassandra codebase that is configurable for each column family/table. The operation of the priority register in the context of the Cassandra workflow complies with the expected behavior that was tested via Thrift as well as the CQL interfaces. The preliminary implementation of the priority register opens various interesting dimensions for the future work including:

Multi-Valued Priority Register As discussed in Section 7.4, the initial implementation of the priority register considers a total order replacement that needs to store only a single value. The future implementation of the priority register will consider a more generic case that needs to store multiple values that are equal in the replacement order.

Adaptive Ordering In the initial implementation of the priority register inside Cassandra, the replacement ordering is set during column family/table creation time. A potential direction for future work is to enable the configuration of the replacement order at runtime via the 'Alter' statement. This would facilitate a potential adaptation of the data replacement logic according to the changing needs of the application.

Application to Content-Based Networking The design of the priority register is motivated by applications in cyber-physical and content-based networking, specifically by SRI's ENCODERS project [WMJ+15]. The adaptation of the priority register to the open-source ENCODERS architecture [enc] and the evaluation of benefits and performance in this application is ongoing work.

Generalization to other Classes of Orders The lexicographical ordering that is used here for the priority register is just one of many variants of application-defined replacement orderings. The applications and feasibility of implementing other types of orderings such as orders induced by information subsumption, information abstraction, or finite partial orders could be studied. As explained earlier, unlike CRDTs, the general concept of partial-order replacement does not come with a fixed reconciliation operation such as the LUB. Hence, if a conflict cannot be resolved directly by the partial order, all data objects involved remain accessible to the application, which can itself trigger a resolution by injecting a new data object that is higher in the replacement order other than the objects in conflict. Clearly, this can be the LUB, but it would also be interesting to identify applications for which a reconciliation via LUB is not appropriate or sufficient.

Portability to other Datastores The concept of the priority register has been implemented and tested for a column family datastore, namely Cassandra. The portability

and ease of implementation of the priority register for other datastore variants such as key-value stores and document datastores would be worth to investigate. We also believe that the integration of more general notions of partial-order replacement into distributed databases is interesting in its own right, because of its applications as a general model for loosely-coupled distributed computing [SKT14].

Release to the Open Source Community The implementation of the priority register is under extensive testing and validation before we plan to release the new module to the Apache Cassandra Open Source Community.

Chapter 8

Conclusion and Future Works

Most of the modern distributed database systems rely on quorum-based replica control protocol and ensures consistency guarantees of the system based on the intersection property of the read and write quorums. One of the limitations of this approach is that the number of nodes to contact before returning response to the client increases linearly depending on the number of replica nodes configured for the data. In addition to an extra communication cost spent on request latency to ensure data consistency, if sufficient number of nodes is unable to contact, the system fails the request affecting the system availability. This thesis addresses this challenge by proposing a new consistency protocol called LibRe, which ensures better tradeoff between request latency and data consistency.

The main goal of the LibRe protocol is to find a middle ground between the default eventual consistency option and the stronger consistency options derived from the quorum intersection property. LibRe is an acronym for Library for Replication. As the name suggests, the protocol maintains a library for each data item whose recent update is not propagated to all the replica nodes. According to the protocol, during write operations, the system issues success message to the client as soon as write is successfully accomplished on one of the replica nodes similar to the eventual consistency option. But unlike eventual consistency option, under LibRe protocol, the replica node sends an *advertisement message* to update an in-memory data structure called the LibRe Registry. By forwarding the read requests to the right replica node by referring to the LibRe Registry, the system reduces the stale read probability of the read operations. Since the consistency guaran-

tees of the LibRe protocol depend on the correctness of the LibRe Registry, one of the main challenges of the protocol is to ensure the correctness of the registry information. The performance of the LibRe protocol was evaluated in maintaining the registry both in centralized as well as in the distributed setup.

Another important problem addressed in this thesis is the challenges involved in evaluating the performance of different consistency options. Evaluating the performance of a distributed systems policy or strategy is expensive in terms of both cost and time. Simulation results are very useful in this situation to obtain an initial performance results. Most of the existing simulation tools for cloud based environment are focused on a coarse-grained problems and do not provide functionalities for simulating application specific semantics. As ensuring replicated data consistency includes application-specific constraints, the existing simulation tools are not well suited for evaluating consistency policies. The thesis work addresses this issue by contributing an open-source simulation toolkit called 'Simizer'. Simizer provides a simple programming interface for defining distributed application behavior that helps to specify how the system should respond according to the incoming requests. By taking the description about the capacity of each node in the cluster and about request pattern as input, simizer evaluates the application performance and provides the needed metrics in an output file.

Using Simizer, we have evaluated the performance of the LibRe protocol against some of the well-known consistency protocols. Following the encouraging results obtained via simulations, the performance of LibRe protocol was tested by distributing the registry information among each node in the cluster. A prototype implementation of the enhanced distributed version of LibRe protocol was developed inside Cassandra distributed storage system as an additional module and is named CaLibRe. The name CaLibRe symbolizes *Cassandra with LibRe*. The performance of the LibRe implementation inside Cassandra (CaLibRe) was benchmarked against Cassandra's native consistency option *one*, *all* and *quorum*. Since existing benchmark tools do not provide needed metrics to evaluate consistency options, one of the existing benchmark tool YCSB (Yahoo! Cloud Serving Benchmark) was extended to evaluate the number of stale reads returned by a storage system under different consistency options. Both simulation and benchmark results confirm that

using LibRe protocol, application would experience similar number of stale reads and availability as that of the stronger consistency options with lower request latency.

Although most of the eventually consistent systems tune the consistency guarantees of the system according to the application or users' decision, deciding an appropriate consistency option of a query/data item during application development time remains difficult for the application developers. In order to take advantage of adaptive consistency feature to a higher extent, the consistency needs of a query/data item has to be chosen during query time rather than during application development. There are enough works in the literature that study the different factors that could influence the consistency needs of a query/data item. Most of these works use a separate service on top of the eventually consistent data store that helps to detect an appropriate consistency option for the incoming requests based on the absorbed factors. However, querying the data store via a centralized service would create a bottleneck for the data store and affects the scalability and throughput of the system. Our thesis work addresses this issue by enabling the data store to adapt the consistency option of the incoming queries according to an external input. The external input could be given by the database administrator or by an external service. Based on the given input, the system can adapt the consistency options of the incoming queries by itself without the intervention of an external service during query time. The proposed model was implemented inside Cassandra and successfully tested for a use case where the load on the system is higher during a specific time-period (peak time) when compared to rest of the time. The use case chosen for our experimentation is the bike sharing system of Paris, Velib'. Since the stronger consistency options need to contact sufficient number of nodes in order to ensure data consistency, applying stronger consistency options during peak time where the load on the system is already high could overload the system. By using LibRe protocol for such use case, an application would experience similar number of stale reads and availability as that of the stronger consistency options while mitigating the load on the system.

Another dimension of this thesis is focused on the Write-Write Inconsistencies that are popularly known as Update-Conflicts. In eventually consistent data stores, where any data replicas of a data item could handle the write operations, the system would often

run-down into Update Conflicts. In order to resolve update conflicts, the system has to do data reconciliation. One of the widely used techniques for resolving update conflicts is via a system actor (user/ application) with the help of a Multi-Value Register. Multi-Value Register keeps all the possible values of the conflicting updates and returns all the possible values during read operations. Following that, a system actor does the reconciliation according to the domain-specific knowledge and writes the resolved value back to the data store. The overhead of this approach includes extra bandwidth and latency to return all the conflicting values during get operations, which could affect the Service Level Agreement (SLA). In addition, if the system user is directly involved in the data reconciliation, there is a cost on the 'bad' user experience that could affect the service popularity of the system. In other case, if the application takes care of the conflict reconciliation without user intervention, additional efforts during the application development are needed.

The contribution of this thesis for this issue is to parameterize a data type using an application-defined replacement ordering. Application-defined replacement orderings specify a necessary condition under which one data item could be replaced by another data item of the same class. A proof-of-concept of such data type named Priority Register was implemented inside the Cassandra distributed data store. The behavior of Priority Register confirms the expected output and validates a data type parameterized by an application-defined replacement ordering would help to move the client-side domain-specific reconciliation directly on the database side.

Perspectives

In this thesis we have addressed different challenges related to quorum-based replica control protocols. The future perspectives of this thesis and open room for the enhancements are summarized as follows.

Generalization and Application of Priority Register for different classes of replacement-orderings: In the prototype implementation of the priority register, the tie between the data items that are equal in the ordering are broken using the timestamp associated with each data item. However, in a general case, the data items those

are equal in the ordering have to be kept at the database side opening room for the additional reconciliation at the client-side. Future perspectives of the priority register has to address this case. Moreover, the replacement ordering that we focused in the thesis is Lexicographical Order, which is just one of many variants of the replacement ordering. The generalization of the priority register model for the application and the feasibility of implementation for the remaining variants of replacement orderings such as information subsumption, information abstraction, or finite partial orders remain open for the future works.

Enhancement and Application of LibRe Protocol: In this thesis, the performance of LibRe protocol was tested in a single data center replication when there is no churn in the system. The performance of the protocol during nodes churn and the effect of the multi-datacenter replication needs to be evaluated. The main goal of LibRe protocol is to find a middle ground between the default eventual consistency and the strong consistency options derived via quorum intersection property. Although the choice between Consistency and Availability conjectured by CAP Theorem is not a binary choice and could be scalable to different intermediate levels, the consistency guarantees of quorum-based replica control protocols remain a binary choice. Quorum-based replica control protocols guarantee data consistency only if a sufficient number of replica nodes are alive, else the protocol fails the request. Hence, it would be helpful to explore more on the middle ground protocols that ensure data consistency with similar latency and availability guarantees as that of the default eventual consistency.

Overriding the application-defined consistency options at the data store level during run-time: Enabling the eventually consistent data stores to override the consistency options dictated by the application requests helps to benefit from the adaptive consistency feature to a higher extent without the intervention of an external service during query time. In the prototype implementation, we have considered a simple use case with pre-computed clustering information. Integrating an online computation technique that can computes the appropriate consistency options for the data items and adapts the consistency levels of the application queries incrementally during run-time remains open

for the future perspective. This may need an additional work on the efficient management of the bloom filters at each node in the cluster.

Contribution to Simizer Open Source Code-base: Simizer is an event-driven simulation toolkit written in Java. The code base of simizer is open-source for the public [1] access. Simizer is currently designed for evaluating Read-Read Inconsistency (Replication Inconsistency). The extension to simizer for evaluating the other inconsistency issues such as Write-Write Inconsistency (Update-Conflicts) and Read-Write Inconsistency (Snapshot Isolation) could be interesting. In addition, the simizer cluster description and network simulation class are considered for single datacenter environment. The extension of simizer for simulating data consistency options for multi-datacenter replication with configurable network class could be helpful.

1. `https://github.com/isep-rdi/simizer`

Bibliographie

[ABA10] DANIEL ABADI. Problems with cap, and yahoo's little known nosql system. http://dbmsmusings.blogspot.fr/2010/04/problems-with-cap-and-yahoos-little.html, April 2010.

[Aba12] Daniel J. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[ACHM11a] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[ACHM11b] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January, 2011, Online Proceedings*, pages 249–260, 2011.

[AD76] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[AEA] D. Agrawal and A. El Abbadi. The generalized tree quorum protocol: An efficient approach for managing replicated data. *ACM Trans. Database Syst.*, 17(4):689–717.

[Bas08] Robert Basmadjian. *UN PROTOCOLE CONTROLE DE REPLIQUE D'UNE STRUCTURE D'ARBORESCENCE ARBITRAIRE - AN ARBITRARY TREE-STRUCTURED REPLICA CONTROL PROTOCOL.* Thése de doctorat, Université Paul Sabatier, Toulouse, France, décembre 2008. (Soutenance le 04/12/2008).

[BCC+03] William H. Bell, David G. Cameron, Luigi Capozza, A. Paul Millar, Kurt Stockinger, and Floriano Zini. Optorsim - a grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing Applications*, page 2003, 2003.

[BHG87] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[BLM+12] Laurent Bobelin, Arnaud Legrand, David Alejandro Gonzàlez Màrquez, Pierre Navarro, Martin Quinson, Frédéric Suter, and Christophe Thiery. Scalable multi-purpose network representation for large scale distributed system simulation. In *Proceedings of the 12th IEEE International Symposium on Cluster Computing and the Grid*, CCGrid'12. IEEE Computer Society Press, 2012.

[BLPF15] Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. Technical report, Microsoft Research, 2015.

[BMM02] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[BP88] Brian N. Bershad and C. Brian Pinkerton. Watchdogs: Extending the unix file system. In *USENIX Winter*, pages 267–275. USENIX Association, 1988.

[BS10] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first*

annual ACM SIGMM conference on Multimedia systems, MMSys '10, pages 35–46, New York, NY, USA, 2010. ACM.

[Bur14] Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1-2):1–150, October 2014.

[BVF+12] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012.

[Cas13] Planet Cassandra. Composite keys in apache cassandra. `http://planetcassandra.org/blog/composite-keys-in-apache-cassandra/`, May 2013.

[Cas14] Planet Cassandra. Wide rows in cassandra cql. `http://planetcassandra.org/blog/wide-rows-in-cassandra-cql`, January 2014.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, 2006.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems (3rd Ed.): Concepts and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CF14] Yousra Chabchoub and Christine Fricker. Classifications of the vélib stations using kmeans, dynamic time wraping and dba averaging method. IWCIM '14. IEEE Computer Society, Nov 2014.

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[CIAP12] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and Maria S. Perez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, CLUSTER '12, pages 293–301, Washington, DC, USA, 2012. IEEE Computer Society.

[CLQ08] Henri Casanova, Arnaud Legrand, and Martin Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society.

[CM86] Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst.*, 4(4):338–378, September 1986.

[CMKS14] Jong-Seok Choi, Tim McCarthy, Minyoung Kim, and Mark-Oliver Stehr. Adaptive wireless networks as an example of declarative fractionated systems. In Ivan Stojmenovic, Zixue Cheng, and Song Guo, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, volume 131 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 549–563. Springer International Publishing, 2014.

[CMY+13] Jong-Seok Choi, T. McCarthy, M. Yadav, Minyoung Kim, C. Talcott, and E. Gressier-Soudan. Application patterns for cyber-physical systems. In *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2013 IEEE 1st International Conference on*, pages 52–59, Aug 2013.

[CPAB13] Houssem-Eddine Chihoub, Maria Perez, Gabriel Antoniu, and Luc Bouge. Chameleon: Customized application-specific consistency by means of behavior modeling. `https://hal.inria.fr/hal-00875947/file/chameleon.pdf`, 2013. [Research Report] <hal-00875947>.

[Cql12] Cql3: allow definition with only a pk. `https://issues.apache.org/jira/browse/CASSANDRA-4361`, 2012.

[CRB+11a] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, C&#x00e9 ;sar A. F. De Rose, and Rajkumar Buyya. Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1):23–50, January 2011.

[CRB+11b] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar A. F. De Rose, and Rajkumar Buyya. Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience (SPE)*, January 2011.

[CRS+08] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo !'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[Dat12] DataStax. Introduction to composite columns. `http://www.datastax.com/dev/blog/introduction-to-composite-columns-part-1`, January 2012.

[dd15] Project Voldemort A distributed database. Configuration. `http://www.project-voldemort.com/voldemort/configuration.html`, September 2015.

[DGMS85] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: A survey. *ACM Comput. Surv.*, 17(3):341–370, September 1985.

[DHJ+07a] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[DHJ+07b] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[Doc15a] DataStax Documentation. Cassandra glossary: upsert. `http://docs.datastax.com/en/cassandra/2.0/share/glossary/gloss_upsert.html`, June 2015.

[Doc15b] DataStax Documentation. Snitches. `http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html`, July 2015.

[EGKM04] Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulieacute ;. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, May 2004.

[enc] ENCODERS. `http://encoders.csl.sri.com/`.

[enc14] ENCODERS software design description v.2.0. `http://encoders.csl.sri.com/wp-content/uploads/2014/08/CBMEN-SRI-Design-Description-V2.0-Dist-A.pdf`, 2014.

[ES83] Derek L. Eager and Kenneth C. Sevcik. Achieving robustness in distributed

database systems. *ACM Trans. Database Syst.*, 8(3):354–381, September 1983.

[Fei02] Dror G. Feitelson. Workload modeling for performance evaluation. In MariaCarla Calzarossa and Salvatore Tucci, editors, *Performance Evaluation of Complex Systems: Techniques and Tools*, volume 2459 of *Lecture Notes in Computer Science*, pages 114–141. Springer Berlin Heidelberg, 2002.

[FFH12] Florian Fittkau, Sören Frey, and Wilhelm Hasselbring. Cdosim: Simulating cloud deployment options for software migration support. In *Proceedings of the 6th IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2012)*, pages 37–46. IEEE Computer Society, September 2012. doi: 10.1109/MESOCA.2012.6392599.

[FGC+97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. *SIGOPS Oper. Syst. Rev.*, 31(5):78–91, October 1997.

[FGJ+78] Michael J. Flynn, Jim Gray, Anita K. Jones, Klaus Lagally, Holger Opderbeck, Gerald J. Popek, Brian Randell, Jerome H. Saltzer, and Hans-Rüdiger Wiehle, editors. *Operating Systems, An Advanced Course*, London, UK, UK, 1978. Springer-Verlag.

[Fid88] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[Geo11] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 1 edition, 2011.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[GJTP12] Katja Gilly, Carlos Juiz, Nigel Thomas, and Ramon Puigjaner. Adaptive

admission control algorithm in a qos-aware web system. *Inf. Sci.*, 199:58–77, September 2012.

[GL02a] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[GL02b] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[GMAB+83] H. Garcia-Molina, T. Allen, B. Blaustein, R. M. Chilenskas, and D. R. Ries. Data-patch: Integrating inconsistent copies of a database after a partition. In *Proceedings of the 3rd IEEE Symposium on Reliability in Distributed Software and Database Systems*. 1983.

[Gro15] GrockDoc. Read repair. `https://www.grockdoc.com/cassandra/2.1/articles/read-repair_69792070-ab3b-4c0c-a0ed-01cb4898e183`, September 2015.

[Hag] Haggle - a content-centric network architecture for opportunistic communication. https://code.google.com/p/haggle/.

[Hal10] Coda Hale. You can't sacrifice partition tolerance. `http://codahale.com/you-cant-sacrifice-partition-tolerance/`, October 2010.

[HC09] Pat Helland and David Campbell. Building on quicksand. *CoRR*, abs/0909.1788, 2009.

[Her86] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comput. Syst.*, 4(1):32–53, February 1986.

[Hew10] Eben Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.

[HHB02] Abdelsalam Helal, Abdelsalam Heddaya, and Bharat K. Bhargava. *Replication Techniques in Distributed Systems*, volume 4 of *Advances in Database Systems*. Kluwer, 2002.

[How93] J.H. Howard. Using reconciliation to share files between occasionally connected computers. In *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pages 56–60, Oct 1993.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[IOY+11] A. Iosup, S. Ostermann, M.N. Yigitbasi, R. Prodan, T. Fahringer, and D. H J Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):931–945, 2011.

[JRS11] F.P. Junqueira, B.C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 245–256, 2011.

[Kan14] C.Y. Kan. *Cassandra Data Modeling and Analysis*. Packt Publishing, 2014.

[KBAK10] D. Kliazovich, P. Bouvry, Y. Audzevich, and S.U. Khan. Greencloud: A packet-level simulator of energy-aware cloud computing data centers. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5, 2010.

[KGK+13] Minyoung Kim, Ashish Gehani, Je-Min Kim, Dawood Tariq, Mark-Oliver Stehr, and Jin-soo Kim. Maximizing availability of content in disruptive environments by cross-layer optimization. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 447–454, New York, NY, USA, 2013. ACM.

[KHAK09] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann.

Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.

[KKS+12] Jinwoo Kim, Minyoung Kim, Mark-Oliver Stehr, Hyunok Oh, and Soonhoi Ha. A parallel and distributed meta-heuristic framework based on partially ordered knowledge sharing. *Journal of Parallel and Distributed Computing*, 72(4):564 – 578, 2012.

[KLCG15] Sathiya Prabhu Kumar, Sylvain Lefebvre, Raja Chiky, and Eric Gressier-Soudan. Calibre: A better consistency-latency tradeoff for quorum based replication systems. In Qiming Chen, Abdelkader Hameurlain, Farouk Toumani, Roland Wagner, and Hendrik Decker, editors, *Database and Expert Systems Applications - 26th International Conference, DEXA 2015, Valencia, Spain, September 1-4, 2015, Proceedings, Part II*, volume 9262 of *Lecture Notes in Computer Science*, pages 491–503. Springer, 2015.

[KLCS14] S.P. Kumar, S. Lefebvre, R. Chiky, and E.G. Soudan. Evaluating consistency on the fly using ycsb. In *IWCIM, 2014.*, pages 1–6, Nov 2014.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[Klo10] Rusty Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[KS93a] P. Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pages 66–70, Oct 1993.

[KS93b] P. Kumar and M. Satyanarayanan. Supporting application-specific resolution

in an optimistically replicated file system. In *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pages 66–70, Oct 1993.

[KSKH10] Minyoung Kim, Mark-Oliver Stehr, Jinwoo Kim, and Soonhoi Ha. An application framework for loosely coupled networked cyber-physical systems. In *IEEE/IFIP Int. Conf. Embedded and Ubiquitous Computing*, EUC'10, pages 144–153, 2010.

[Kum91] Akhil Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. Comput.*, 40(9):996–1004, September 1991.

[Kum15] Sathiya Prabhu Kumar. Evaluating data consistency using ycsb. `https://github.com/isep-rdi/YCSB`, September 2015.

[LAEA95a] M. L. Liu, D. Agrawal, and A. El Abbadi. The performance of replica control protocols in the presence of site failures. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, SPDP '95, pages 470–, Washington, DC, USA, 1995. IEEE Computer Society.

[LAEA95b] M.L. Liu, D. Agrawal, and A. El Abbadi. The performance of replica control protocols in the presence of site failures. In *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, pages 470–477, Oct 1995.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, C-28(9):690–691, Sept 1979.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, January 1989.

[LAS13] Rui Liu, Ashraf Aboulnaga, and Kenneth Salem. Dax: A widely distributed

multitenant storage service for dbms hosting. *Proc. VLDB Endow.*, 6(4):253–264, February 2013.

[Lef13] Sylvain Lefebvre. *Load distribution services for the Cloud : a multimedia data management example.* Theses, Conservatoire national des arts et metiers - CNAM, December 2013.

[LKC14] Sylvain Lefebvre, Sathiya Prabhu Kumar, and Raja Chiky. Simizer: Evaluating consistency trade offs through simulation. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 6:1–6:2, New York, NY, USA, 2014. ACM.

[LLJ07] Yijun Lu, Ying Lu, and Hong Jiang. Idea:: An infrastructure for detection-based adaptive consistency control in replicated services. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 223–224, New York, NY, USA, 2007. ACM.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[LPC+12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[LS] Kapang Lego and Dipankar Sutradhar. Comparative study of adhoc routing protocol aodv, dsr and dsdv in mobile adhoc network 1.

[M-.08] M-.O. Stehr and Carolyn Talcott. Planning and learning algorithms for routing in disruption-tolerant networks. In *In Proc. of IEEE Military Communications Conference, MILCOM 2008*, 2008.

[Mai08] Siddharth Maini. Mobile database systems, vijay, kumar. wiley-interscience inc., hoboken, nj (2006), isbn: 0-471-46792-8. *Inf. Process. Manage.*, 44(1):405–407, 2008.

[MB02] Manzur Murshed and Rajkumar Buyya. Using the gridsim toolkit for enabling grid computing education. In *Proc. of the Int. Conf. on Communication Networks and Distributed Systems Modeling and Simulation*, pages 18–24, 2002.

[MFF] S. Mccanne, S. Floyd, and K. Fall. ns2 (network simulator 2). http://www-nrg.ee.lbl.gov/ns/.

[Mol07] Ingo Molnár. Cfs scheduler, 2007.

[MR97] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. STOC '97, pages 569–578. ACM, 1997.

[MRW97] Dahlia Malkhi, Michael Reiter, and Rebecca Wright. Probabilistic quorum systems. PODC '97, pages 267–273. ACM, 1997.

[MSV+10] Jesús Montes, Alberto Sánchez, Julio J. Valdés, María S. Pérez, and Pilar Herrero. Finding order in chaos: A behavior model of the whole grid. *Concurr. Comput. : Pract. Exper.*, 22(11):1386–1415, August 2010.

[MSZ11] Carlos Baquero Marc Shapiro, Nuno Preguicça and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. RR-7506, INRIA, 2011.

[Mur13] Arinto Murdopo. Consistency tradeoff in modern distributed db. http://www.otnira.com/2012/04/21/consistency-tradeoff-in-modern-distributed-db/, June 2013.

[Net] Networked Cyber-Physical Systems @ SRI. `http://ncps.csl.sri.com`.

[NVPC+12] Alberto Núñez, Jose L. Vázquez-Poletti, Agustin C. Caminero, Gabriel G. Castañé, Jesus Carretero, and IgnacioM. Llorente. icancloud: A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1):185–209, 2012.

[NW98] Moni Naor and Avishai Wool. The load, capacity, and availability of quorum systems. *SIAM J. Comput.*, 27(2):423–447, April 1998.

[OO15] Diego Ongaro and John Ousterhout. The raft consensus algorithm. https://raftconsensus.github.io, June 2015.

[OUMI06] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p collaborative editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, CSCW '06, pages 259–268, New York, NY, USA, 2006. ACM.

[PBA+10] Nuno M. Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Dotted version vectors: Logical clocks for optimistic replication. *CoRR*, abs/1011.5808, 2010.

[PMSL09] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 395–403, Washington, DC, USA, 2009. IEEE Computer Society.

[PPR+83] Jr. Parker, D.S., Gerald J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *Software Engineering, IEEE Transactions on*, SE-9(3):240–247, May 1983.

[Pri08] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[PSL03] Nuno Preguiça, Marc Shapiro, and J. Legatheaux Martins. Automating semantics-based reconciliation for mobile transactions. pages 515–524, La-Colle-sur-Loup, France, October 2003.

[Pun94] Kumar Puneet. *Mitigating the effects of optimistic replication in a distributed file system.* PhD thesis, Carnegie-Mellon University. Computer Science Department. CMU-CS-94-215., 1994.

[Ram03] Gehrke Ramakrishnan. *Database Management Systems.* McGraw-Hill, third edition, 2003.

[RD15a] Riak-Doc. Riak 2.0. http://docs.basho.com/riak/latest/intro-v20/, June 2015. Version 2.1.1.

[RD15b] Riak-Docs. Client and server side conflict resolution. http://docs.basho.com/riak/latest/dev/using/conflict-resolution/#Client-and-Server-side-Conflict-Resolution, June 2015. Version 2.1.1.

[RD15c] Riak-Docs. Read repair vs active anti entropy. https://docs.basho.com/riak/2.1.1/theory/concepts/aae/#Read-Repair-vs-Active-Anti-Entropy, June 2015. Version 2.11.

[RHR+94] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 12–12, Berkeley, CA, USA, 1994. USENIX Association.

[RJKL11] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3):354–368, March 2011.

[RL93] Michael Rabinovich and Edward D. Lazowska. An efficient and highly available read-one write-all protocol for replicated data management. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, PDIS '93, pages 56–66, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[Sco12] Colin Scott. Latency numbers every programmer should know. http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html, 2012.

[SE98] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.

[SHCD06] James Scott, Pan Hui, Jon Crowcroft, and Christophe Diot. Haggle: A networking architecture designed around mobile users. In *Proceedings of the Third Annual IFIP Conference on Wireless On-Demand Network Systems and Services (WONS 2006)*. IEEE, January 2006.

[She15] Justin Sheehy. There is no now: Problems with simultaneity in distributed systems. `https://queue.acm.org/detail.cfm?id=2745385`, March 2015.

[Sit08] Emil Sit. *Storing and Managing Data in a Distributed Hash Table*. PhD thesis, Massachusetts Institute of Technology, June 2008.

[SKG+12] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[SKRC10] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

[SKS06] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.

[SKT10] Mark-Oliver Stehr, Minyoung Kim, and Carolyn Talcott. Toward distributed declarative control of networked cyber-physical systems. In *Proc. 7th Int. Conf. Ubiquitous Intelligence and Computing*, UIC'10, pages 397–413. Springer-Verlag, 2010.

[SKT14] Mark-Oliver Stehr, Minyoung Kim, and Carolyn Talcott. Partially ordered knowledge sharing and fractionated systems in the context of other models for distributed computing. In *Specification, Algebra, and Software*, volume 8373 of *Lecture Notes in Computer Science*, pages 402–433, 2014.

[SL13a] Georgia Sakellari and George Loukas. A survey of mathematical models, simulation approaches and testbeds used for research in cloud computing. *Simulation Modelling Practice and Theory*, 2013.

[SL13b] Raja Chiky Sylvain Lefebvre, Sathiya Prabhu.K. Simizer: A cloud simulation tool. https://forge.isep.fr/projects/simizer/, March 2013.

[SPBZ11a] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, January 2011.

[SPBZ11b] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[SSH+07] Jing Su, James Scott, Pan Hui, Jon Crowcroft, Eyal De Lara, Christophe Diot, Ashvin Goel, Meng How Lim, and Eben Upton. Haggle: Seamless networking for mobile applications. In *Proceedings of the 9th International Conference on Ubiquitous Computing*, UbiComp '07, pages 391–408, Berlin, Heidelberg, 2007. Springer-Verlag.

[SZWL11] S. Sakr, Liang Zhao, H. Wada, and A. Liu. Clouddb autoadmin: Towards a truly elastic cloud-based data store. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 732–733, July 2011.

[TAPV10] M. Tlili, R. Akbarinia, E. Pacitti, and P. Valduriez. Scalable p2p reconciliation infrastructure for collaborative text editing. In *Advances in Databases Knowledge and Data Applications (DBKDA), 2010 Second International Conference on*, pages 155–164, April 2010.

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[TTP+95] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5):172–182, December 1995.

[Vel] Velib - mairie de paris. http://en.velib.paris.fr.

[Vog08] Werner Vogels. Eventually consistent. http://queue.acm.org/detail.cfm ?id=1466448, October 2008.

[Vog09] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[Vog12] Werner Vogels. Back-to-basics weekend reading - weighted voting for replicated data. `http://www.allthingsdistributed.com/2012/11/weighted-voting.html`, November 2012.

[Vol15] Project Voldemort. Physical architecture options. `http://www.project-voldemort.com/voldemort/design.html`, April 2015.

[Vuk10] Marko Vukolic. Remarks: The origin of quorum systems. *Bulletin of the EATCS*, 102:109–110, 2010.

[VV15] Paolo Viotti and Marko Vukolic. Consistency in non-transactional distributed storage systems. *CoRR*, abs/1512.00168, 2015.

[WFZ+11] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 134–143. www.cidrdb.org, 2011.

[Wik13a] Cassandra Wiki. Architectureantientropy. `https://wiki.apache.org/cassandra/ArchitectureAntiEntropy`, November 2013.

[Wik13b] Cassandra Wiki. Cassandra limitations. `http://wiki.apache.org/cassandra/CassandraLimitations`, November 2013.

[Wik13c] Cassandra Wiki. Hinted handoff. `https://wiki.apache.org/cassandra/HintedHandoff`, November 2013.

[WMJ+15] Samuel Wood, James Mathewson, Joshua Joy, Mark-Oliver Stehr, Minyoung Kim, Ashish Gehani, Mario Gerla, Hamid Sadjadpour, and J.J. Garcia-Luna-Aceves. ICEMAN: A practical architecture for situational awareness at the network edge. In *Logic, Rewriting, and Concurrency*, Lecture Notes in Computer Science, 2015.

[WPE+83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. *SIGOPS Oper. Syst. Rev.*, 17(5):49–70, October 1983.

[WYW+10] Ximei Wang, Shoubao Yang, Shuling Wang, Xianlong Niu, and Jing Xu. An application-based adaptive replica consistency for cloud storage. In *Proceedings of the 2010 Ninth International Conference on Grid and Cloud Computing*, GCC '10, pages 13–17, Washington, DC, USA, 2010. IEEE Computer Society.

[XSK+14] Chao Xie, Chunzhi Su, Manos Kapritsos, Yang Wang, Navid Yaghmazadeh, Lorenzo Alvisi, and Prince Mahajan. Salt: Combining acid and base in a distributed database. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 495–509, Berkeley, CA, USA, 2014. USENIX Association.

[YV00] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 21–21, Berkeley, CA, USA, 2000. USENIX Association.

[ZWXY13] Hao Zhang, Yonggang Wen, Haiyong Xie, and Nenghai Yu. *Distributed Hash Table - Theory, Platforms and Applications.* Springer Briefs in Computer Science. Springer, 2013.

Glossaire

— $CNAM$: Conservatoire National des Arts et Métiers

— *LibRe*: Library for Replication.

— *CaLibRe*: Cassandra with LibRe.

— $YCSB$: Yahoo Cloud Serving Bechmark.

— $ACID$: Atomicity, Consistency, Isolation, Durability

— $BASE$: Basic Availability, Soft state, Eventual consistency.

— DHT: Distributed Hash Table.

— $ROWA$: Read One, Write All.

— $ROWA - A$: Read One, Write All Available.

le cnam

Sathiya Prabhu KUMAR

Adaptive Consistency Protocols for Replicated Data in Modern Storage Systems with High Degree of Elasticity

le cnam

Résumé :
Cette thèse adresse le domaine scientifique des systèmes distribués et plus particulièrement de la cohérence des données répliquées partagées pour les systèmes de stockage de nouvelle génération. Ce travail de thèse aborde trois défis qui contribuent à la gestion de la cohérence des données pour des systèmes fondés sur la cohérence éventuelle (eventual Consistency). La première contribution porte sur LibRe, "Library for Replication", un protocole de gestion de cohérence des données centré sur les lectures qui minimise la latence des opérations de lecture et d'écriture. LibRe est réalisé sous la forme d'une bibliothèque. Afin d'évaluer la performance de LibRe, nous avons contribué à l'implantation d'un outil de simulation sous licence logiciel libre, Simizer, et, à l'extension du benchmark YCSB. La seconde contribution vise à optimiser la gestion de la cohérence des données à l'exécution. Au lieu de choisir une option de cohérence par défaut lors des accès, le système peut la remplacer grâce à des informations externes, fournies par un contexte, une application, un administrateur. Ce modèle est validé par une implémentation au sein de Cassandra. Enfin, la dernière contribution se concentre sur le problème de la réconciliation de données lors de l'émergence de copies incohérentes. Elle s'appuie sur la notion de Registre à Priorité (Priority Register). Elle permet de déplacer la résolution de la réconcilation du client vers le serveur de base de données. La réconciliation peut alors s'appuyer sur des critères spécifiques à l'application.
Mots clés :
NoSql, Théorème CAP, Cohérence éventuelle, Systèmes de Quorums.

Abstract :
The contribution of this thesis work mainly focuses on three types of challenges in ensuring data consistency of an eventually consistent data store. In the first contribution, the thesis focuses on ensuring consistency of the read operation with minimum read and write latencies with the help of a proposed protocol called LibRe. LibRe is an acronym for Library for Replication. According to the protocol, the system keeps track of data items whose recent update is not propagated to all the replica nodes. The registry is used during read operations to forward the read requests to a replica node that contains the most recent version of the needed data item. In order to evaluate the performance of LibRe protocol, the thesis also contributed implementation of an open-source simulation toolkit called Simizer and the extension of an existing benchmark tool called YCSB. The second contribution of the thesis tries to eliminate the necessity to contact a middleware service for choosing needed consistency options of the application queries during run-time. The final contribution focuses on moving slow and tedious client-side data reconciliation process directly on the database-side with the help of a proposed data type called Priority Register.
Keywords :
NoSql Systems, CAP Theorem, Eventual Consistency, Quorum Systems.