HAL
open science

A Machine-Checked Proof of Correctness of Pastry

Noran Azmy

To cite this version:

Noran Azmy. A Machine-Checked Proof of Correctness of Pastry. Networking and Internet Architecture [cs.NI]. Université de Lorraine, 2016. English. NNT : 2016LORR0277 . tel-01558422

HAL Id: tel-01558422

https://theses.hal.science/tel-01558422

Submitted on 7 Jul 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE LORRAINE

AVERTISSEMENT

Ce document est le fruit d'un long travail approuvé par le jury de soutenance et mis à disposition de l'ensemble de la communauté universitaire élargie.

Il est soumis à la propriété intellectuelle de l'auteur. Ceci implique une obligation de citation et de référencement lors de l'utilisation de ce document.

D'autre part, toute contrefaçon, plagiat, reproduction illicite encourt une poursuite pénale.

Contact : ddoc-theses-contact@univ-lorraine.fr

LIENS

Code de la Propriété Intellectuelle. articles L 122. 4
Code de la Propriété Intellectuelle. articles L 335.2- L 335.10
http://www.cfcopies.com/V2/leg/leg_droi.php
http://www.culture.gouv.fr/culture/infos-pratiques/droits/protection.htm

SAARLAND UNIVERSITY

COMPUTER SCIENCE

A Machine-Checked Proof of Correctness of Pastry

Thesis for obtaining the title of Doctor of Natural Science of the Faculty of Natural Science and Technology I of Saarland University

Noran Azmy

Saarbrücken / July 2016

École doctorale IAEM Lorraine

Une preuve certifiée par la machine de la correction du protocole Pastry

THÈSE

présenté et soutenue le 31 juillet 2016

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Noran Azmy

Composition du jury

Examinateurs : Christoph Weidenbach
Stephan Merz
Olivier Perrin
Heike Wehrheim
Frank-Olaf Schreyer

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503
Max Planck Institute for Informatics
Saarland University

Dean:	Prof. Dr. Frank-Olaf Schreyer
Date:	24 November 2016

Defense-Committee:

Chair:	Prof. Bernd Finkbeiner, Ph.D.
Reviewers:	Prof. Dr. Christoph Weidenbach
	Prof. Dr. Stephan Merz
	Prof. Dr. Olivier Perrin
	Prof. Dr. Heike Wehrheim
Academic Assistant:	Dr. Andreas Nonnengart

Résumé

Les réseaux pair-à-pair (P2P) constituent un modèle de plus en plus populaire pour la programmation d'applications Internet car ils favorisent la décentralisation, le passage à l'échelle, la tolérance aux pannes et l'auto-organisation. à la différence du modèle traditionnel client-serveur, un réseau P2P est un système réparti décentralisé dans lequel tous les nœuds interagissent directement entre eux et jouent à la fois les rôles de fournisseur et d'utilisateur de services et de ressources. Une table de hachage distribuée (DHT) est réalisée par un réseau P2P et offre les mmes services qu'une table de hachage classique, hormis le fait que les différents couples (clef, valeur) sont stockés dans différents nœuds du réseau. La fonction principale d'une DHT est la recherche d'une valeur associée à une clef donnée. Parmi les protocoles réalisant une DHT on peut nommer Chord, Pastry, Kademlia et Tapestry. Ces protocoles promettent de garantir certaines propriétés de correction et de performance ; or, les tentatives de démontrer formellement de telles propriétés se heurtent invariablement à des cas limites dans lesquels certaines propriétés sont violées. Tianxiang Lu a ainsi décrit des problèmes de correction dans des versions publiées de Pastry. Il a conçu un modèle, appelé LuPastry, pour lequel il a fourni une preuve partielle, mécanisée dans l'assistant à la preuve TLA$^+$ Proof System, démontrant que les messages de recherche de clef sont acheminés au bon nœud du réseau dans le cas sans départ de nœuds. En analysant la preuve de Lu j'ai découvert qu'elle contenait beaucoup d'hypothèses pour lesquelles aucune preuve n'avait été fournie, et j'ai pu trouver des contre-exemples à plusieurs de ces hypothèses. La présente thèse apporte trois contributions. Premièrement, je présente LuPastry$^+$, une spécification TLA$^+$ révue de LuPastry. Au-delà des corrections nécessaires d'erreurs, LuPastry$^+$améliore LuPastryen introduisant de nouveaux opérateurs et définitions, conduisant à une spécification plus modulaire et isolant la complexité de raisonnement à des parties circonscrites de la preuve, contribuant ainsi à automatiser davantage la preuve. Deuxièmement, je présente une preuve TLA$^+$ complète de l'acheminement correct dans LuPastry$^+$. Enfin, je démontre que l'étape finale du processus d'intégration de nœuds dans LuPastry(et LuPastry$^+$) n'est pas nécessaire pour garantir la cohérence du protocole. Concrètement, j'exhibe une nouvelle spécification avec un processus simplifié d'intégration de nœuds, que j'appelle Simplified LuPastry$^+$, et je démontre qu'elle garantit le bon acheminement de messages de recherche de clefs. La preuve de correction pour

Simplified LuPastry$^+$est obtenue en réutilisant la preuve pour LuPastry$^+$, et ceci représente un bon succès pour la réutilisation de preuves, en particulier considérant la taille de ces preuves. Chacune des deux preuves requiert plus de 30000 étapes interactives ; à ma connaissance, ces preuves constituent les preuves les plus longues écrites dans le langage TLA$^+$ à ce jour, et les seuls exemples d'application de preuves mécanisées de théorèmes pour la vérification de protocoles DHT.

Abstract

A distributed hash table (DHT) is a peer-to-peer network that offers the function of a classic hash table, but where different key-value pairs are stored at different nodes on the network. Like a classic hash table, the main function provided by a DHT is key lookup, which retrieves the value stored at a given key. Examples of DHT protocols include Chord, Pastry, Kademlia and Tapestry. Such DHT protocols certain correctness and performance guarantees, but formal verification typically discovers border cases that violate those guarantees. In his PhD thesis, Tianxiang Lu reported correctness problems in published versions of Pastry and developed a model called LuPastry, for which he provided a partial proof of correct delivery of lookup messages assuming no node failure, mechanized in the TLA$^+$ Proof System. In analyzing Lu's proof, I discovered that it contained unproven assumptions, and found counterexamples to several of these assumptions. The contribution of this thesis is threefold. First, I present LuPastry$^+$, a revised TLA$^+$ specification of LuPastry. Aside from needed bug fixes, LuPastry$^+$ contains new definitions that make the specification more modular and significantly improve proof automation. Second, I present a complete TLA$^+$ proof of correct delivery for LuPastry$^+$. Third, I prove that the final step of the node join process of LuPastry/LuPastry$^+$ is not necessary to achieve consistency. In particular, I develop a new specification with a simpler node join process, which I denote by Simplified LuPastry$^+$, and prove correct delivery of lookup messages for this new specification. The proof of correctness of Simplified LuPastry$^+$ is written by reusing the proof for LuPastry$^+$, which represents a success story in proof reuse, especially for proofs of this size. Each of the two proofs amounts to over 32,000 proof steps; to my knowledge, they are currently the largest proofs written in the TLA$^+$ language, and—together with Lu's proof—the only examples of applying full theorem proving for the verification of DHT protocols.

Zusammenfassung

Eine verteilte Hashtabelle (DHT) ist ein P2P Netzwerk, das die gleiche Funktion wie eine klassische Hashtabelle anbietet, wo aber verschiedene Schlüssel-Inhalt Paare an verschiedenen Knoten im Netzwerk gespeichert werden. Chord, Pastry, Kademlia und Tapestry sind einige bekannte Implementierungen von DHT. Solche Protokolle versprechen bestimmte Eigenschaften bezüglich Korrektheit und Leistung. Die formale Verifikation von diesen Protokollen führt jedoch normalerweise zu Widersprüchen dieser Eigenschaften. In seiner Doktorarbeit entdeckt Tianxiang Lu Gegenbeispiele zu veröffentlichten Versionen von Pastry und entwickelt LuPastry, ein Pastry Model ausschließlich des Knotenausfalles. Zusätzlich bietet Lu einen Teilbeweis für korrekte Lieferung von Suchnachrichten in LuPastry in der Sprache TLA$^+$ an. Lus Beweis basiert auf unbewiesenen Annahmen. Beim Untersuchen des Beweises habe ich Gegenbeispiele zu mehreren dieser Annahmen entdeckt. Diese Doktorarbeit deckt drei Hauptthemen ab. Erstens, es wird LuPastry$^+$ entwickelt: eine revidierte TLA$^+$ Spezifikation zu LuPastry. Neben den benötigten Fehlerkorrekturen, bietet LuPastry$^+$ zusätzlich neue Definitionen an, welche die Spezifikation modularer machen, und die Automatisierung des Beweises signifikant verbessern. Zweitens, biete ich einen vollständigen TLA$^+$ korrektheitsbeweis für LuPastry$^+$ an. Drittens, zeige ich, dass der letzte Schritt des Beitrittsprotokolles in LuPastry/LuPastry$^+$ nicht notwendig für Korrektheit ist. Insbesondere, biete ich eine neue Spezifikation mit einem einfacheren Beitrittsprotokoll an, und einen Korrektheitsbeweis dafür. Nach bestem Wissen sind diese Beweise (2 Beweise je von über 32.000 Schritten) bis dato die größten in TLA$^+$ geschriebenen Beweise.

Acknowledgments

Christoph Weidenbach and Stephan Merz have been the best mentors anyone could hope for—supportive in difficult times, hard on me when I needed it, and extremely informative when I had no clue. Uwe Waldmann's classes initiated me into the field, and he continued to offer me a helping hand throughout my master's and Ph.D. My office mates Daniel Wand and Mathias Fleury as well as the rest of the RG1 and VeriDis teams made the workplace a fun place to be. Special thanks to Tianxiang Lu, whose work is the basis of this dissertation, and Hernán Vanzetto, who dedicated a large portion of his busy schedule for me in my early days. I would not have been here without the Max Planck Institute's funding and support; a big, warm thanks to everyone who makes education possible. I would also not have been here but for my family; my warm, loud, bona fide Egyptian family where everyone takes a surprising interest in your proof of correctness of a P2P protocol. From my mom, also moonlighting as my rock, my dad, a.k.a. provider of all my comforts, my brilliant little sister, my lighthouse, who serves dutifully as my big sister when necessary, her husband and my new brother whose love for her makes her shine even brighter, to my aunt and grandmothers who are buckets of kindness and delight in the shape of human beings. Also belonging to this category, under the title "Brother from Another Mother", is my best friend Eman, who taught me more about myself than I could have ever learned on my own. I have been equally blessed in friends as in family: Nahla, a.k.a. "Partner in Crime" and "Most Selfless Human Being"; Rana, who has been there for me every day of a six-year journey of ups and big downs; Hagar, my role model and powerhouse; Areej, the genuine heart who sees me as I want to be seen and not as I really am, and her husband Eslam who has been there for me through thick and thin; My Powerpuff Girls Iva and Maha, Maha, you push me to be a better person and I want to make you proud; Iva, your friendship has been the most important pillar in my life the past years and I will not try to do it justice here; Yusra, you are a lesson on kindness and awesomeness; Dahlia, you're beautiful inside and out; and finally, to Mohammed and Walaa, my home, my conscience and my mirror. My final thanks goes to technology, to people who made the internet what it is and made all the things that make life and work easier, to people who support and donate to these things. Wikipedia, online open course-ware, my Spotify study playlist; these things remind me everyday why I am in tech and that ideas can change the world.

Contents

Contents **viii**

List of Figures **ix**

1 Introduction **1**

2 Introduction **13**

3 Preliminaries **25**
3.1 Peer-to-Peer (P2P) Networks 25
3.2 Pastry 27
3.3 The TLA$^+$ Language 30

4 LuPastry$^+$ **39**
4.1 An Overview of the LuPastry$^+$ Model 39
4.2 The TLA$^+$ Specification of LuPastry$^+$ 43

5 Improvements in LuPastry$^+$ to LuPastry **61**
5.1 Lu's Partial Proof of Correct Delivery 61
5.2 Improvements to the LuPastry Specification 65
5.3 LuPastry$^+$ Proof Structure 72

6 The LuPastry$^+$ Correctness Proof **75**
6.1 Intuition 78
6.2 Assumptions and Arithmetic Axioms 82
6.3 Correctness Invariants 84
6.4 An Outline of the Reduction Proof 99

7 Simplified LuPastry$^+$ **103**
7.1 A Simplified Join Process for LuPastry$^+$ 103
7.2 The TLA$^+$ Specification of Simplified LuPastry$^+$ 105
7.3 Correct Delivery in Simplified LuPastry$^+$ 110

8 Conclusion **113**

Bibliography **117**

List of Figures

2.1 The TLA$^+$ proof system. 22
2.2 Methodology used for the formal verification of Pastry using TLA$^+$. 23

3.1 The client/server model versus the decentralized P2P model. . . . 26
3.2 An example distributed hash table with four nodes and 256 keys. . 28
3.3 A summary of TLA$^+$ syntax. 33
3.4 TLA$^+$ specification of the "Two Jugs" problem. 36
3.5 TLA$^+$ example proof for the "Two Jugs" problem. 37

4.1 An example of a Pastry ring of size 16, and the ideal distribution of coverage among its live nodes. 40
4.2 An example routing table of a node with ID 10233102, where node IDs are interpreted as 8 digits in base $2^B = 4$. 41
4.3 The LuPastry/LuPastry$^+$ join protocol. 44
4.4 An overview of the LuPastry$^+$ specification in TLA$^+$. 45

5.1 Structure of the original LuPastry specification and proof. 62
5.2 Analysis of Lu's unproven assumptions about the leaf set data structure. 64
5.3 Structure of the original LuPastry proof versus the new, complete proof. 73

6.1 Structure of the LuPastry$^+$ correctness proof. 77
6.2 Possible arrangements for two Ready nodes i and j and their coverage ranges. 78
6.3 Network stability. 82
6.4 Possible arrangement for Ready/OK nodes i and k with respect to a key j right-covered by i. 101

7.1 The Simplified LuPastry$^+$ join protocol. 104

Chapitre 1

Introduction

Les réseaux pair-à-pair (P2P) constituent un modèle qui est de plus en plus utilisé pour la programmation d'applications Internet modernes. à la différence du modèle traditionnel client-serveur qui repose sur un serveur central fournissant des ressources aux clients, un réseau P2P est un système distribué et décentralisé dans lequel tous les nœuds (aussi appelés *pairs*) interagissent directement entre eux et jouent à la fois les rles de fournisseurs et d'utilisateurs de services et de ressources.

Le modèle P2P a connu sa première heure de gloire en 1999 avec la popularité du système Napster de partage de fichiers qui permettait à ses utilisateurs de chercher et de télécharger des fichiers mis à disposition par d'autres utilisateurs [23]. Cependant, Napster peut être considéré comme un réseau P2P de *première génération* qui nécessitait encore un serveur central d'indexe contenant des informations sur les utilisateurs et sur leurs contenus partagés. Ce serveur était responsable pour la recherche de fichiers, alors que l'échange de fichiers était réalisé directement entre deux utilisateurs. Plus tard, les services évoluaient vers un modèle tout à fait décentralisé, à la base d'applications comme Gnutella et Kazaa qui fonctionnaient en l'absence de tout serveur central. Ces applications sont souvent appelées des applications P2P de *seconde génération*.

Les réseaux P2P favorisent le passage à l'échelle et la robustesse car ils ne nécessitent pas de serveur central qui représente un point unique de défaillance ou encore un goulot d'étranglement pour la performance. Ils sont aussi moins coteux et plus faciles à déployer car ils s'auto-organisent et ne nécessitent pas d'équipements spécifiques concernant les serveurs ou les systèmes d'exploitation. Ceci étant dit, certaines applications P2P à large échelle optent pour une architecture semi-distribuée en utilisant un ou plusieurs nœuds spéciaux, parfois appelés *super-nœuds*, qui sont toujours en ligne et garantissent ainsi au réseau une connectivité et fiabilité accrues.

Aujourd'hui la technologie P2P se trouve dans des applications très diverses, allant du partage de fichiers au visionnage de contenus multi-média et à la téléphonie par Internet. Ainsi Skype, l'application populaire de téléphonie

audio et vidéo, repose-t-il sur un réseau P2P entièrement décentralisé, ainsi qu'un certain nombre de super-nœuds. Le site Web officiel indique que chaque nouveau nœud ajouté au réseau contribue à la puissance de traitement et à la bande passante. Ainsi, en décentralisant les ressources, les réseaux P2P de seconde génération ont réussi à quasiment éliminer les cots associés à une infrastructure centralisée importante [14]. Le site Web argumente aussi que le modèle P2P correspond au modèle prévu à l'origine pour le Web, dans lequel les utilisateurs contribuent de nouveaux contenus (des pages Web) et accèdent aux contenus créés par d'autres utilisateurs.

Un exemple supplémentaire d'une application P2P populaire est le *flash player* d'Adobe Systèmes, un des visionneurs de médias le plus utilisés sur Internet, qui repose désormais sur la technologie P2P pour visionner du contenu en temps réel et sur demande. Adobe indique qu'une page Web fournissant de l'audio et vidéo à votre ordinateur peut livrer le contenu avec une meilleure performance si les utilisateurs qui visionnent le même contenu partagent leur bande passante. De cette manière, l'audio ou vidéo peut être servie de manière plus lisse, sans coupures ou pauses dues à l'approvisionnement. Ceci est appelé du réseautage assisté par les pairs car les pairs sur le réseau assistent les uns les autres afin de fournir une meilleure expérience [22].

D'autres applications reposaient sur la technologie P2P à un certain moment mais l'abandonnaient plus tard en faveur du modèle centralisé. Pendant un certain temps, la British Broadcasting Corporation (BBC) utilisait un protocole P2P pour soutenir son iPlayer BBC bien connu, une application qui permettait aux utilisateurs de visionner des vidéos de la BBC. Dans un entretien datant de 2008 avec Anthony Rose du groupe *BBC Controller Vision and Online Media*, celui-ci affirmait que deux ans auparavant l'utilisation de P2P offrait des bénéfices clairs et substantiels mais que l'abandon de cette technologie était justifié par les baisses dramatiques du prix de la bande passante qui rétablissaient la faisabilité du modèle client-serveur [36]. Rose se réservait également la possibilité d'un retour ultérieur au P2P. Le service Spotify de *streaming* audio faisait aussi appel à la technologie P2P pour son client sur ordinateur jusqu'en 2014 quand il revenait à un modèle client-serveur, pour des raisons similaires [39]. Enfin, la plate-forme Livestation pour la distribution en temps réel de la télévision et de la radio utilisait également une technologie P2P rachetée à Microsoft Research.

L'application qui est probablement la plus fortement associée avec la technologie P2P dans l'opinion publique est BitTorrent, un protocole de communication pour l'échange P2P de fichiers par Internet. En effet, en date de 2013 le trafic BitTorrent à lui seul est responsable de 3,35% de la bande passante utilisée par toutes les applications Internet dans le monde [34]. Les utilisateurs installent un *client* BitTorrent au choix parmi plusieurs disponibles, alors qu'un *tracker* BitTorrent fournit une liste de fichiers disponibles pour l'échange et assiste le client établir une communication avec d'autres pairs qui possèdent le fichier. Il s'agit d'un modèle proche de celui de Napster dans lequel le tracker représente un serveur central qui initie la communication parmi les pairs mais o ceux-ci continuent à interagir de manière décentralisée sans

l'aide du serveur. Cependant, de nombreux clients BitTorrent actuels utilisent un système sans tracker dans lequel chaque pair agit comme tracker, résultant en un modèle entièrement décentralisé. De nombreuses entreprises distribuant du logiciel ou du multi-média se servent de la technologie BitTorrent pour offrir des téléchargements. Cette méthode de distribution est particulièrement recommandée par d'importants projets du logiciel libre comme la distribution Linux Ubuntu[1] ou la suite bureautique LibreOffice[2], afin d'améliorer la disponibilité des téléchargements et de réduire la charge des serveurs propres à ces projets.

Un problème majeur concernant les réseaux P2P à large échelle purs, c'est à dire entièrement décentralisés, réside dans la gestion efficace des ressources disponibles. Un réseau P2P non-structuré sans contrainte sur la topologie des nœuds participants est très inefficace : une fonction comme la recherche d'un fichier donné demanderait dans un tel réseau d'inonder le réseau avec une requête de recherche jusqu'à ce que cette requête atteigne un pair possédant le fichier demandé. Ceci donnerait lieu à une quantité importante de trafic réseau inutile, ainsi qu'à des charges importantes d'utilisation de la CPU et de la mémoire.

Une *table de hachage distribuée* (distributed hash table, DHT) offre un moyen de structurer un réseau P2P de manière à organiser les ressources disponibles, et de rendre les communications entre les pairs fiables et efficaces. Une DHT est un réseau P2P qui sert de table de hachage, répartissant les couples (clef, valeur) parmi les différents nœuds du réseau. Les nœuds qui forment la DHT se voient affectés des identifiants uniques, et les messages entre les nœuds sont acheminés par une route impliquant des nœuds intermédiaires et déterminée en fonction de l'identifiant du nœud cible, plutt que par inondation du réseau. Comme une table de hachage classique, une DHT offre deux fonction principales : $put(k, v)$ associe la valeur v à la clef k, alors que $get(k)$ récupère la valeur associée à la clef k. En raison de l'absence de serveur central ayant une vue globale du réseau, les nœuds d'une DHT doivent collaborer pour déterminer l'ensemble de clefs stockées à chaque nœud, ainsi que pour acheminer les requêtes *put* et *get* au nœud concerné.

Les tables de hachage distribuées bénéficient des avantages combinés des réseaux P2P et des tables de hachage, de par une conception simple et élégante permettant la localisation d'une donnée demandée avec une très bonne efficacité, et ne nécessitant aucune information globale. Dans la plupart des applications pratiques, les réseaux P2P/DHT sont sujets à un taux important de *churn*, des nœuds rejoignant et quittant le réseau en permanence, ainsi qu'à des défaillances spontanées de nœuds qui partent sans en informer au préalable d'autres nœuds du réseau. L'implémentation d'une DHT doit être capable de gérer ces turbulences de manière efficace et gracieuse, assurant une reconfiguration du réseau vers un état stable qui garantit une connectivité entre les nœuds vivants et un consensus sur l'ensemble de clefs affectées à chaque nœud.

[1] `www.ubuntu.com`
[2] `www.libreoffice.org`

La dernière décennie a vu de grands efforts de recherche dans le domaine des réseaux P2P, et en particulier des DHT. De nombreuses implémentations de DHT ont été proposées dans la littérature scientifique, et beaucoup de travaux ont été entrepris afin d'étudier les propriétés de ces implémentations et comment les améliorer. Parmi les protocoles DHT les plus connus figurent Pastry, Chord, Kademlia et CAN [37, 38, 32, 35]. Ces protocoles sont similaires dans le sens qu'ils se focalisent sur la gestion efficace de données stockées de manière répartie parmi un grand nombre de nœuds. Cependant, ils diffèrent par rapport à des caractéristiques telles que la topologie du réseau logique, la fonction pour calculer la distance entre deux nœuds du réseau, et le modèle de routage. Par exemple, Chord organise les nœuds dans un cercle virtuel appelé l'anneau Chord. Pastry repose sur une structure arborescente de nœuds, assistée par une structure annulaire similaire à celle de Chord et qui sert de base au routage quand la structure arborescente est incapable de déterminer la cible appropriée. Dans Chord, ainsi que dans d'autres protocoles, les nœuds exécutent périodiquement un algorithme de stabilisation, en échangeant des messages de *ping* afin de maintenir à jour leurs informations locales concernant le réseau. L'article [46] donne un excellent aperçu des différentes variantes de DHT, leurs théories sous-jacentes, et leurs applications.

Alors que les entreprises optant pour la technologie P2P utilisent généralement des implémentations propriétaires, comme dans les exemples cités précédemment, ces implémentations reposent souvent sur des protocoles proposés dans la littérature scientifique. Un exemple bien connu d'applications fondées sur des DHT est le tracking réparti des systèmes BitTorrent sans *trackers* explicites mentionnés plus haut qui est implanté par de nombreux clients BitTorrent. Ainsi, l'implémentation Mainline DHT de tracking réparti pour BitTorrent, fondée sur Kademlia, est probablement l'implémentation la plus importante de DHT en pratique. Une étude menée en 2013 estimait le nombre d'utilisateurs de Mainline DHT entre 10 et 25 millions, avec un *churn* quotidien d'au moins 10 millions [40]. Des implémentations propriétaires de DHT incluent Oracle Coherence, la grille de données résidant dans la mémoire vive d'Oracle implantée en Java et le service DynamoDB de bases de données proposé par Amazon.

Les publications concernant les protocoles DHT typiquement affirment certains taux de stabilité et de fiabilité de ces protocoles en fonction du *churn*. Cependant, ces affirmations ne sont généralement fondées que sur des démonstrations papier, au mieux. Observant les problèmes de connectivité de certaines applications P2P populaires soumises au *churn*, on s'est rendu compte au cours de ces dernières années que la technologie DHT actuelle ne peut pas garantir des niveaux de fiabilité tels qu'imaginés à l'origine. Par exemple, l'application Skype fondée sur une implantation P2P qui est très certainement une DHT, a subi deux coupures massives en 2007 et en décembre de 2010, cette dernière étant due à une défaillance spontanée d'un nombre important de super-nœuds. Dans quelques heures à peine, le nombre d'utilisateurs connectés à Skype tomba de 23,3 millions à environ 1,6 millions, et la coupure dura deux jours.

De telles coupures représentent bien évidemment une facture importante pour les entreprises. Ce risque, couplé à une utilisation en augmentation de

la technologie P2P/DHT dans de nombreuses applications demandant de la tolérance aux pannes, a motivé des travaux de recherche s'intéressant à la vérification de protocoles P2P, et particulièrement DHT, en employant des méthodes formelles telles que la modélisation formelle, la simulation et la vérification automatique par model checking. Naturellement, l'analyse et la vérification de tels systèmes répartis de large échelle sont tout sauf triviales. En effet, en raison de la taille et de la complexité de tels systèmes, les efforts de vérification ont le plus souvent été limités à l'étude de certains fragments de ces protocoles – tels que la recherche de clefs dans un réseau statique sans arrivée ou départ de nœuds – et ils ont typiquement fait appel à des méthodes légères telles que la modélisation et la simulation.

La présente thèse s'insère dans cette ligne de recherche : elle propose une étude de la correction de Pastry sur la base de la preuve de théorèmes mécanisée. Plus précisément, je vérifie le protocole de la recherche de clefs (requêtes *get*) pour deux variantes de Pastry en donnant des preuves de correction complètes et rigoureuses dans l'assistant interactif à la preuve TLA$^+$. Il a déjà été démontré dans [27] qu'aucune version publiée de Pastry ne garantit cette propriété de correction car des ajouts et départs de nœuds peuvent causer une séparation irréversible du réseau, conduisant à un acheminement erroné de messages de recherche de clefs. Dans cette thèse je démontre que le modèle de Pastry sans départ de nœuds garantit l'acheminement correct de messages de recherche de clefs. Je montre également que, toujours dans le modèle sans départ de nœuds, la correction est maintenue par un processus d'ajout de nœuds qui est plus simple que ceux proposés pour Pastry, par exemple dans les articles [21, 27].

Dans ce qui suit, je discute certains travaux connexes traitant du sujet de la vérification formelle de protocoles P2P. Ensuite, je décris la contribution de cette thèse à ce domaine de recherche. Ceci est suivi d'une courte description des outils et techniques utilisés dans cette thèse, et en particulier de l'assistant à la preuve TLA$^+$. Enfin, je donne un court résumé de la suite de la thèse.

Travaux connexes

Un grand nombre d'études scientifique a été dédié pendant ces dernières années au protocole de DHT Chord qui présente de fortes similarités à Pastry. Il implante une table de hachage distribuée de la manière suivante. Une fonction fixe de hachage associe à chaque clef et à chaque nœud vivant de Chord un identifiant de m bits (l'identifiant d'un nœud est calculé en appliquant la fonction de hachage à son adresse IP alors que l'identifiant d'une clef est déterminée en hachant la clef). L'espace d'identifiants est assimilé à un cercle appelé l'anneau Chord qui contient les 2^m identifiants possibles entre 0 et $2^m - 1$. La clef k est associée au premier nœud n dont l'identifiant est égal à celui de k ou qui le suit dans le sens de l'anneau ; n est appelé le *nœud successeur* de k. Chaque nœud n maintient une liste de ses successeurs dans l'anneau, ainsi qu'un *tableau de doigts* (*finger table*) pour un routage efficace, ce dernier contenant

les adresses d'au plus m nœuds à différents catégories de distance de n. Plus spécifiquement, la i-ème entrée dans cette table contient le premier nœud à une distance d'au moins 2^{i-1} identifiants de n sur l'anneau. Les nœuds de Chord utilisent un algorithme périodique de stabilisation pour maintenir à jour leurs listes de successeurs et tableaux de doigts. Un routage correct des requêtes est garanti si chaque nœud connat son nœud successeur immédiat sur l'anneau. En présence de défaillances de nœuds, cette hypothèse ne peut être maintenue à tout moment, et l'effet de défaillances de nœuds est mitigé par la réplication, conduisant au maintien de listes de successeurs : plus grande la taille de la liste de successeurs et plus grande sera la probabilité que la requête d'une clef renvoie le nœud correct.

Bakhshi et al. [4] décrivent un modèle abstrait de réseaux P2P structurés en anneau dans le π-calcul. Ils utilisent ce modèle pour vérifier l'algorithme de stabilisation de Chord en exhibant une bisimulation faible entre la spécification de Chord en tant que réseau en anneau et l'implémentation de l'algorithme de stabilisation. Cette étude est menée dans un modèle d'ajout pur de nœuds mais sans défaillances, et des entités comme les tableaux de doigts ou listes de successeurs ne sont pas prises en compte.

Zave a mené beaucoup de travaux au sujet de la vérification de Chord. Utilisant Alloy pour modéliser formellement et pour vérifier Chord, elle démontre que le modèle d'ajout pur – c'est à dire dans lequel aucun nœud ne part du réseau – est correct mais que le protocole complet n'assure pas forcément les invariants affirmés [44]. Dans l'article [45] elle présente une version de Chord assortie d'une preuve partiellement mécanisée de correction.

$\mathcal{DKS}$ [1] est une implémentation d'une table de hachage distribuée dont le nom est dérivée de *recherche k-aire répartie*. $\mathcal{DKS}(N, k, f)$ décrit un protocole P2P pour un réseau de taille maximale N, une arité de recherche de k dans le réseau et un paramètre f de tolérance aux pannes. Brièvement, la recherche k-aire répartie garantit que n'importe quelle clef t peut être localisée dans au plus $\log_k(N)$ sauts. Initialement, l'espace de recherche est identique à l'espace entier de clefs dans la table de hachage. à chaque pas de la recherche distribuée, l'espace de recherche actuel est divisé en k parties de tailles égales, dont chacune est sous la responsabilité d'un nœud différent. En répétant ces sous-divisions jusqu'à ce que chaque partie ne contienne qu'un élément, l'algorithme aboutit à la partie contenant la clef t qui correspond à la cible de la recherche.

$\mathcal{DKS}$ peut être vu comme une généralisation de systèmes comme Chord, Pastry ou Tapestry. Comme Chord, $\mathcal{DKS}$ repose sur une structure en anneau, bien que les auteurs affirment que cette structure soit arbitraire et que les algorithmes proposés fonctionnent également pour d'autres topologies de réseaux. Le point distinctif de $\mathcal{DKS}$ est que, contrairement à d'autres protocoles, il ne requiert pas de routine corrective pour maintenir à jour des tableaux de routage ou d'autres informations locales. En effet, les tableaux de routage sont corrigés à la volée pendant les opérations de recherche et d'insertion. Cette idée repose sur l'observation que, dans les systèmes P2P o le nombre de recherches et insertions est beaucoup plus grand que celui d'ajouts de nœuds, de départs ou de défaillances, la correction des tableaux de routage à la volée pendant

que le protocole effectue des opérations de recherche et d'insertion évite une consommation inutile de la bande passante due à un mécanisme périodique de correction.

$\mathcal{DKS}$ permet aux nœuds de rejoindre ou de quitter le réseau à n'importe quel moment. Comme dans le cas de Chord, un nœud rejoint un réseau $\mathcal{DKS}$ en envoyant une requête correspondante à son successeur sur l'anneau. Rejoindre l'anneau est un événement atomique dans le sens qu'un nœud i recevant plus d'une requête à rejoindre l'anneau ne peut en traiter plus d'une à la fois. Un nœud peut soit quitter l'anneau de manière gracieuse en informant d'autres nœuds de son intention de partir, soit il peut échouer abruptement. Il est affirmé que le système continue à fournir un service fiable de recherche de clefs pourvu que pas plus de f nœuds consécutifs n'échouent en même temps.

Les auteurs de $\mathcal{DKS}$ mènent des expériences par simulation afin de déterminer l'effet du *churn* sur l'efficacité du traitement des requêtes de clefs [1]. Le protocole $\mathcal{DKS}$ est décrit en détail dans la thèse de Ghodsi [20]. Dans cette thèse, Ghodsi discute plusieurs problèmes tels que des demandes d'intégration à l'anneau concurrentes à des défaillances de nœuds, et il affirme qu'on ne peut pas garantir la correction en présence de défaillances arbitraires de nœuds, à cause d'une possible séparation du réseau. Toujours concernant $\mathcal{DKS}$, citons le travail de Borgstrm et al. [9] qui utilise CCS pour la vérification formelle du protocole de recherche de clefs dans le cas statique, c'est à dire sans la prise en compte de l'arrivée ou du départ de nœuds.

Le travail le plus proche, et qui constitue également le point de départ de cette thèse, a été mené par Lu [29, 30, 27] concernant Pastry. Pastry [37] est un réseau structuré en anneau, de manière similaire à Chord, et chaque nœud maintient des listes de successeurs et de prédecesseurs, appelées l'ensemble de feuilles (*leaf set*) du nœud. Au lieu du tableau de doigts de Chord, chaque nœud dispose d'un tableau de routage dont les entrées sont déterminées par la longueur du préfixe de bits commun entre les identifiants de l'entrée et du nœud. Comme dans le cas de Chord, l'acheminement correct de requêtes de clefs peut être garanti par Pastry si chaque nœud a un ensemble de feuilles correct, ou au moins connat ses successeur et prédecesseur sur l'anneau, indépendamment du contenu du tableau de routage.

Lu modélise Pastry en TLA$^+$, et il utilise l'outil TLC de model checking, ainsi que l'assistant TLAPS à la preuve, pour vérifier formellement le bon acheminement de de requêtes de clefs (messages *get*) : *à tout moment il existe au plus un nœud qui puisse répondre à une requête de recherche de clef, et ce nœud est le nœud vivant le plus proche à la clef.* Comme dans le cas de Chord, Lu découvre plusieurs problèmes dans le protocole Pastry original, même après avoir pris en compte des améliorations proposées. Enfin il présente une variante de Pastry appelée LuPastryet ne prenant en compte que l'intégration de nœuds, et il vérifie l'acheminement correct sous l'hypothèse forte qu'aucun nœud ne fait défaillance. Il est à noter que la variante de Pastry proposée par Lu repose aussi sur des ajouts atomiques de nœuds dans le sens qu'un nœud vivant est seulement autorisé à assister un seul nœud à la fois à rejoindre l'anneau, comme dans le cas de $\mathcal{DKS}$. Dans sa preuve, Lu réduit la propriété du

bon acheminement à un ensemble d'environ 50 invariants que le protocole est affirmé vérifier, et ces invariants sont démontrés à l'aide de TLAPS. Dans ce sens, LuPastryreprésente un effort majeur dans le domaine de la vérification formelle mécanisée d'algorithmes répartis.

Néanmoins, la preuve de Lu repose sur de nombreuses hypothèses présentées sans preuve concernant l'arithmétique et les structures de données spécifiques au protocole. En examinant la preuve, j'ai découvert des contre-exemples à plusieurs de ces hypothèses de base. Alors que j'ai pu démontrer des variantes plus fortes de plusieurs de ces hypothèses, ceci n'a pas été possible pour d'autres. En effet, j'ai pu trouver un contre-exemple à l'un des invariants affirmés par Lu, la preuve duquel n'a été possible grce à des hypothèses incorrectes. Cette observation m'a conduit à concevoir une nouvelle preuve de correction pour LuPastry, et ce faisant à améliorer la spécification TLA$^+$ du protocole dans plusieurs respects. Au cours de mon travail de preuve j'ai observé que le sous-protocole d'intégration de nœuds peut être simplifié de manière significative sans en affecter la correction. Plus précisément, l'étape finale appelée échange de bail (*lease exchange*) que le nouveau nœud doit effectuer avec ses nœuds voisins avant de participer activement au protocole n'est pas nécessaire pour démontrer la correction dans les scénarios o il n'y a que des ajouts de nœuds au réseau. En effet, cet échange de bail ne faisait pas partie du protocole d'origine publié dans [37] mais a seulement été introduit par les auteurs dans un article ultérieur, parmi d'autres améliorations au protocole. Bien qu'aucune raison explicite n'ait été donnée, on peut considérer que la raison probable à l'ajout de cette étape était d'améliorer la justesse des ensembles de feuilles et, par conséquent, la fiabilité de la recherche de clefs. Comme démontré par Lu, le protocole entier dans lequel des nœuds peuvent arriver et partir librement ne garantit pas le bon acheminement de messages de recherche de clefs, même en présence d'échange de bail. Par contre, j'observe qu'en absence de départ de nœuds, le protocole est correct même sans cette étape. à la suite de cette observation, j'ai formalisé une variante plus simple de la spécification de LuPastryqui s'affranchit de l'étape d'échange de bail, et j'ai démontré que cette variante garantit le bon acheminement de messages.

Contributions de cette thèse

Cette thèse fait les contributions suivantes.

1. **LuPastry$^+$.** Une version améliorée de la spécificaton TLA$^+$ de LuPastry[27] est proposée. Au-delà de la correction d'erreurs, la structure de la spécification est plus modulaire, en introduisant de nouveaux opérateurs qui permettent d'améliorer notablement le degré d'automatisation de la preuve de correction.

2. **Preuve de correction de LuPastry$^+$.** Une preuve complète et rigoureuse en TLA$^+$ du bon acheminement de requêtes de clefs par LuPastry$^+$est présentée.

3. **LuPastry$^+$simplifié.** Une variante de LuPastry$^+$est définie dans laquelle le protocole d'intégration de nœuds est simplifié en omettant l'étape d'échange de bail.

4. **Preuve de correction pour LuPastry$^+$simplifié.** Une preuve complète et rigoureuse en TLA$^+$ du bon acheminement de messages de requêtes de clefs par LuPastry$^+$simplifié est présentée. Cette preuve est une adaptation de la preuve de correction de LuPastry, et représente en cela un bon succès en matière de réutilisation de preuves.

Cette thèse présente une contribution originale dans les sens suivants. D'abord, elle représente une étude de deux variantes dynamiques de Pastry, sans se restreindre à une vue statique du réseau, en permettant à des nœuds d'intégrer le réseau de manière concurrente. Ensuite, la vérification du protocole repose sur la preuve mécanisée de théorèmes plutt que sur des méthodes moins rigoureuses comme la simulation ou le model checking d'instances finies. Enfin, la preuve de correction de LuPastry$^+$simplifié, obtenue en adaptant celle de LuPastry$^+$, représente une réussite unique en matière de réutilisation de preuves, spécialement pour une preuve de taille aussi importante contenant plus de 32000 interactions de preuve.

Outils et techniques

Vérification formelle

La vérification formelle d'un système informatique (matériel ou logiciel) est le fait de s'assurer de la correction du système, c'est à dire que le système fonctionne comme prévu en garantissant un ensemble d'exigences ou propriétés spécifiques, sur la base de méthodes mathématiques formelles.

Il existe différentes approches à la vérification formelle qui varient de par leur rigueur, et par conséquent de par le temps et effort nécessaires à leur application. Quelle que soit l'approche choisie, la première étape consiste en une *spécification* du système : un modèle du système décrivant son comportement attendu, rédigé dans une notation mathématique choisie, qu'elle soit fondée sur la logique, les automates, les réseaux de Petri, les algèbres de processus ou la théorie des ensembles. Une spécification doit être aussi précise, non-ambigue, concise et complète que possible. Cette thèse implique deux techniques de vérification formelle : le model checking et la preuve mécanisée.

Model checking. Par model checking, on entend un ensemble de techniques pour vérifier que toute exécution du système vérifie les propriétés énoncées. Pour des systèmes à nombre d'états fini, cette approche est rigoureuse. Il est également possible d'utiliser le model checking pour des modèles infinis, à condition que des ensembles infinis d'états puissent être représentés de manière finitaire. Pour des systèmes de taille arbitraire, le model checking suffit pour

vérifier des instances du système de taille fixe, et ceci aide à découvrir des erreurs et à gagner en confiance en la correction de la spécification du système entier, mais ne sert pas de preuve dans l'absolu. En d'autres termes, le model checking est utile pour démontrer la présence plutt que l'absence d'erreurs [6]. Model checking présente l'avantage d'être entièrement automatisé et de ne demander que peu d'interaction de la part de l'utilisateur. Cependant, un problème typique avec le model checking est l'explosion d'états car pour la plupart des systèmes pratiques l'espace d'états est soit infini soit crot de manière exponentielle avec le nombre de paramètres variables du système [3].

Preuve mécanisée. La vérification déductive dénote le processus de générer une collection d'*obligations de preuve* mathématiques dont la véracité implique la correction du système. Ces obligations sont alors *déchargées*, ou démontrées, à l'aide d'un outil dédié à la *preuve mécanique*. Un outil automatique de preuve de théorèmes essaie de démontrer ou d'infirmer une obligation de preuve donnée de manière entièrement automatisée, sans interférence de la part de l'utilisateur. De tels outils peuvent être généralistes tels que Spass [41] ou Zenon [8], ainsi que des prouveurs SMT (satisfiabilité modulo théories) tels que CVC4 [5] ou Z3 [16] qui se chargent d'obligations de preuve relatives à une certaine théorie comme l'arithmétique ou la théorie des ensembles.[3] Pour des systèmes de grande taille, un *outil interactif de preuve*, aussi appelé *assistant à la preuve*, est un outil logiciel qui assiste un utilisateur en la rédaction de preuves mathématiques dont les obligations de preuve élémentaires sont déléguées à un outil de preuve automatique. L'application de la preuve de théorèmes à la vérification de systèmes de grande taille peut être un processus long et coteux, et il demande une très fine compréhension du système, ainsi que de nombreuses interactions avec l'assistant à la preuve.

TLA$^+$

TLA$^+$ est un langage de spécification formelle dédié à la modélisation et la vérification de systèmes répartis. TLA$^+$ permet d'écrire une spécification du système, d'en énoncer les propriétés et enfin d'écrire des preuves de ces propriétés.

Le langage de spécification est fondé sur la logique temporelle des actions (*temporal logic of actions*, TLA), une variante de la logique temporelle de temps linéaire définie par Leslie Lamport en 1994, pour la spécification du comportement dynamique du système, et sur la théorie non-typée des ensembles due à Zermelo et Fraenkel pour décrire les structures de données. Les systèmes sont décrits en tant que machines à états opérant sur des n-uplets de variables d'états, en définissant un prédicat d'états *Init* et un prédicat de transitions *Next* qui contraignent respectivement les états initiaux possibles du système

[3] Plus correctement, étant donné que ces deux théories sont très expressives et indécidables, les solveurs SMT se focalisent typiquement sur des fragments décidables de ces théories comme l'arithmétique linéaire sur les entiers ou les réels, sans quantificateurs.

et sa relation de transition. Un prédicat de transition (appelé aussi *action*) est une formule de la logique du premier ordre contenant de variables non-primées et primées qui dénotent respectivement les valeurs de ces variables dans les états avant et après la transition.

Les propriétés énoncées d'une spécification TLA$^+$ sont elles aussi exprimées en tant que formules de la logique temporelle. Ces propriétés peuvent être vérifiées sur des instances finies de la spécification en faisant appel à TLC [43], le model checker par énumération explicite d'états associé à TLA$^+$. Pour des instances de taille infinie ou arbitraire, les propriétés peuvent être démontrées en rédigeant une preuve qui peut être certifiée à l'aide de TLAPS (*TLA$^+$ Proof System*, [15]), dont l'architecture est présentée à la figure 2.1. La base de TLAPS est un langage hiérarchique de preuves : l'utilisateur rédige une preuve TLA$^+$ formée d'une hiérarchie d'*étapes de preuve*. Chaque étape est interprétée par le *gestionnaire de preuves* qui génère des obligations de preuve correspondantes et les envoie à des outils automatiques de preuve incluant Zenon, Isabelle/TLA$^+$ et des solveurs SMT. Des étapes plus difficiles qui ne peuvent être démontrées immédiatement par les outils automatiques peuvent être décomposées en de multiples sous-étapes. à cause de la nature non-typée du langage, une partie de l'effort de preuve consiste en la démonstration d'un invariant de typage qui fixe en particulier les domaines et co-domaines des fonctions et opérateurs.

TLA$^+$ et ses outils sont accessibles depuis l'interface TLA$^+$ toolbox. La toolbox est un environnement intégré de développement qui permet aux utilisateurs de rédiger des spécifications TLA$^+$, d'exécuter le model checker TLC et de rédiger et certifier des preuves en faisant appel au gestionnaire de preuves. Les langages de spécification et de preuves TLA$^+$ sont brièvement expliqués au chapitre 3.

Méthodologie

Je démontre la propriété de sreté qui exprime le bon acheminement des requêtes pour deux variantes différentes de Pastry. La propriété affirme qu'*à tout moment d'une exécution il existe au plus un nœud qui puisse répondre à une requête de clef, et ce nœud est le nœud vivant le plus proche à cette clef* [27].

Pour chaque variante de Pastry, la méthodologie empruntée pour trouver la preuve est illustrée à la figure 2.2. La variante de Pastry en question est décrite par une spécification TLA$^+$ et la propriété de bon acheminement est démontrée en la réduisant à des *invariants inductifs* de la spécification. Une propriété P est un invariant d'une spécification si elle est vraie à tous les états initiaux de la spécification, ainsi qu'à tous les états accessibles à partir des états initiaux. La propriété P est un invariant inductif si elle est vraie à tous les états initiaux de la spécification, et si sa véracité est préservée par toutes les transitions. D'après cette définition tout invariant inductif est un invariant car une propriété vraie à tous les états initiaux et préservée par toutes les transitions est nécessairement vraie à tous les états accessibles. Le plus grand défi dans l'élaboration de la preuve est de trouver les invariants inductifs qui sont suffisants et corrects. Ce processus de découverte des invariants sous-jacents se déroule comme suit.

Je commence par exprimer une propriété P qui m'apparat à la fois comme utile pour la preuve du bon acheminement et comme invariant correct de la spécification. Je me sers du model checking pour vérifier s'il y a des contre-exemples à cette propriété dans un réseau de petite taille, par exemple de quatre ou huit nœuds. Si TLC trouve un contre-exemple, alors P n'est pas un invariant de la spécification, et il est nécessaire de reformuler la propriété de manière appropriée. Si aucun contre-exemple n'est trouvé alors on peut avoir assez de confiance pour procéder à l'étape suivante en démontrant par induction, à l'aide de TLAPS, que P est un invariant inductif. Ce processus peut toujours révéler des contre-exemples à la propriété P qui n'ont pas pu être trouvés par model checking, et dans ce cas la propriété doit toujours être révisée. Si la démonstration établit P comme un invariant de la spécification, alors la propriété est ajoutée à la liste d'invariants qui peuvent être utilisés dans la preuve du bon acheminement. Cette dernière preuve se déroule en parallèle au processus de générer et de modifier des invariants : les invariants découverts sont motivés par la preuve du bon acheminement, et ils influent à leur tour sur cette preuve.

Organisation

Cette thèse est organisée comme suit. Le chapitre 3 contient des notions préliminaires sur les réseaux pair-à-pair, les tables de hachage distribuées, la vérification formelle et le langage et l'assistant à la preuve TLA$^+$. Dans le chapitre 4 je présente le modèle LuPastry$^+$ainsi que sa spécification en TLA$^+$. Dans le chapitre 5 je décris les améliorations que ma nouvelle spécification LuPastry$^+$apporte à la spécification originale de LuPastrypar Lu. Je résume également mon analyse de la preuve partielle de correction par Lu et décris des contre-exemples que j'ai pu découvrir à certaines de ses hypothèses, et à l'un des invariants affirmés dans la preuve. Enfin, j'explique la structure de ma nouvelle preuve rigoureuse de correction pour LuPastry$^+$. Cette preuve elle-même est décrite au chapitre 6. Dans le chapitre 7 je présente le protocole LuPastry$^+$simplifié dans lequel la phase d'échange de bail est éliminée du processus d'intégration de nœuds, et je montre comment j'adapte la preuve TLA$^+$ de LuPastry$^+$pour obtenir une nouvelle preuve de correction du protocole simplifié. Enfin, le chapitre 8 contient des remarques de conclusion et une discussion des limitations de ce travail, ainsi que d'éventuels travaux futurs dans cette direction.

Chapter 2

Introduction

Peer-to-peer (P2P) networks are an increasingly popular model for modern internet applications. In contrast to the traditional client/server model, which relies on a centralized server providing resources to clients, a P2P network is a distributed, decentralized system in which all nodes, or *peers*, communicate directly with each other and act as both providers and users of services and resources.

The P2P model came to popularity with the rise of the file sharing system Napster in 1999, which allowed users to search and download files found on other users' devices [23]. However, Napster can be seen as a *first generation* P2P network, which still relied on a central index server that indexed users and their shared content. Searching for files took place through the server, while file transfer took place directly between two users. Later on, the trend moved towards a completely decentralized model with applications like Gnutella, which could operate without any central servers. These applications are often termed *second generation* P2P applications.

P2P networks are favored for their scalability and robustness, since there is no central server representing a single point of failure or a performance bottleneck. They are also cheaper and easier to set up, since they are self-organized and there is no need for any special server equipment or operating system. That said, some large-scale P2P applications opt for a semi-distributed approach, making use of one or more special nodes, sometimes termed *supernodes*, that remain online to provide the network with better connectivity and reliability.

Today, P2P technology is used in a large variety of applications from file sharing to multimedia streaming and online telephony. Skype, the popular voice and video chatting application, relies on a completely decentralized P2P network with a number of supernodes. According to their official web site, "Each new node added to the network adds potential processing power and bandwidth to the network. Thus, by decentralizing resources, second generation (2G) P2P networks have been able to virtually eliminate costs associated with a large, centralized infrastructure" [14]. It is also argued on the web page that the P2P model was the original model envisioned for the world wide web

at the time of its creation, with users both creating new content (web pages) and accessing content created by other users.

Another example of a popular P2P application is Adobe System's flash player, which is one of the most widely-used media players on the internet, and now based on P2P technology for support of live and on-demand streaming. Adobe states that "A website that serves audio and video to your computer can deliver the content with better performance if users who are playing the same content share their bandwidth. Sharing bandwidth allows the audio or video to play more smoothly, without skips or pauses from buffering. This is called peer-assisted networking, since peers on the network assist each other to provide a better experience" [22].

Other applications relied on P2P technology at some point, but later abandoned it in favor of the centralized model. For some time, the British Broadcast Corporation (BBC) used P2P as the underlying model for its famous BBC iPlayer, a desktop application which allows users to stream BBC videos. In a 2008 interview with Anthony Rose, BBC Controller Vision and Online Media Group, he stated that "there were definite and substantial benefits from using P2P two years ago", but that the shift away from P2P was justified because of the dramatic decline in the price of bandwidth, making the client/server model feasible once more [36]. Rose also expressed the possibility of a return to P2P in the future. Spotify, the music streaming service, also relied on P2P technology for music streaming on their desktop client until they moved back to a client/server model in 2014, citing similar reasons [39]. Livestation, a platform for distributing live television and radio online, also started out based on P2P technology acquired from Microsoft Research.

Perhaps the application that is most strongly associated in the minds of the public with P2P technology is BitTorrent, a communications protocol for P2P file sharing over the internet. In fact, as of 2013, BitTorrent traffic amounts to 3.35% of all internet application bandwidth worldwide [34]. Users install one of the several available BitTorrent *clients*, and a BitTorrent *tracker* provides a list of files available for transfer and helps the client communicate with other peers that have the file. This is a Napster-like model where the tracker represents a central server that initiates the communication between peers, but where peers continue to communicate in a decentralized way without the help of the server. However, many BitTorrent clients now use a *trackerless system*, where every peer acts as a tracker, resulting in a completely decentralized model. Many software and multimedia companies now offer some downloads using BitTorrent technology. It is particularly encouraged as a download method by major open-source and free software projects such as the Linux distribution Ubuntu[1], or the LibreOffice office suite[2], in order to improve download availability and reduce the load on the project's own servers.

A key problem with pure, completely decentralized P2P networks, and in particular large-scale networks, is how to efficiently manage the available re-

[1] `www.ubuntu.com`
[2] `www.libreoffice.org`

sources. A completely unstructured P2P network where no topology is imposed on the participating nodes is highly inefficient, since in such a network, functions like search for a given file would have to resort to flooding the network with a search request until the request reaches a peer that has the required file. Flooding causes a very high amount of unnecessary network traffic and CPU/memory usage [47, 19].

Distributed hash tables (DHT) are a way of structuring P2P networks so that the available resources are organized, and communication among peers is reliable and efficient. A DHT is a P2P network that acts as a hash table where different (key-value) pairs are stored at different nodes on the network. Nodes forming a DHT are assigned unique identifiers, and messages from one node to another—instead of being flooded through the network—are routed through a number of intermediate nodes, the route being determined by node identifiers. Like a classic hash table, a DHT offers two main functions: $put(k, v)$ stores the value v at key k, and $get(k)$ retrieves the value associated with key k. Due to the lack of a central server with a global view of the network, nodes in a DHT must collaborate to decide on their respective key storage range, and to route *put* and *get* requests to the appropriate node.

Distributed hash tables tap the advantages of both P2P communication and hash tables, with a simple and elegant design that enables locating a required piece of data with high efficiency, and without the need for global information. In most practical applications, P2P/DHT networks are subject to a high level of *churn*; nodes are continuously joining and leaving the network, and may fail abruptly without giving notice to other nodes on the network. The DHT implementation should handle this turbulence efficiently and smoothly and ensure that the network always recovers into a stable state where connectivity among the live nodes is maintained and there is no confusion as to which node stores which portion of the key space.

The past decade has seen extensive research in the area of P2P networks, and especially DHT. Numerous implementations of DHT have been proposed in research publications, and a lot of work has been done to study the properties of these implementations and how to improve them. Among the most popular DHT protocols are Pastry, Chord, Kademlia and CAN [37, 38, 32, 35]. These protocols are similar in that they focus on the efficient management of data stored in a distributed fashion over a large number of nodes, but they differ in some characteristics such as the topology of the overlay network, the distance function between nodes on the network, and the routing model. In Chord, for example, nodes are organized in a circle called the Chord ring. Pastry uses a tree-like structure of nodes, aided by a ring structure similar to that of Chord that is used for routing when the tree structure cannot find the proper target. In Chord and other protocols, nodes employ a periodic "stabilization algorithm" by exchanging ping messages to keep their local information about the network up to date. An excellent survey of the different variants of DHT, the underlying theory and applications can be found in [46].

While companies opting for P2P technology typically base their services on their own proprietary implementations as in the examples mentioned above,

these implementations are often based on or similar to the implementations proposed in scientific publications. A prominent example of DHT-based applications is "distributed tracking" for trackerless BitTorrent systems mentioned above, supported by many popular BitTorrent clients. Arguably the largest practical implementation of DHT is the distributed BitTorrent tracking implementation Mainline DHT, based on Kademlia. In 2013, a study measured Mainline DHT users to range from 10 million to 25 million, with a daily churn of at least 10 million [40]. Proprietary DHT implementations include Oracle Coherence, Oracle's Java-based in-memory data grid, and Amazon's DynamoDB database service [17].

Published proposals of DHT protocols typically make claims regarding the reliability and stability of the given protocol under churn. However, these claims are usually supported by paper-and-pencil arguments at best. With popular P2P-based applications suffering from connectivity problems due to churn, it has become increasingly clear over the years that the reliability of current DHT technology cannot be guaranteed as was once believed. Skype, for example, which is based on a proprietary P2P implementation that is most likely a DHT, has suffered two major outages in 2007 and December of 2010, the latter being due to the abrupt failure of a large number of "supernodes". In just a few hours, the number of online Skype users dropped from 23.3 million users to around 1.6 million. The outage lasted for two days.

Such outages obviously come with a huge bill for business. This risk, along with the increasing usage of P2P/DHT technology in many applications where fault-tolerance is required, has motivated a line of research that looks into the verification of P2P protocols, and in particular DHT, using formal methods like formal modeling, simulation and model checking. Naturally, the analysis and verification of such large distributed systems is anything but trivial. In fact, due to the size and complexity of such systems, most verification efforts have been limited to studying only fragments of these protocols—such as lookups in a static network where nodes do not join or leave—typically using lightweight methods such as modeling and simulation.

This thesis is a work in this line of research that studies the correctness of Pastry using full theorem proving. I verify the correctness of lookups (*get* requests) for two variants of Pastry by giving a complete proof of correctness using the interactive proof assistant TLA$^+$. It has already been shown in [27] that all published versions of Pastry violate this correctness property, since node leaves and failures may cause the network to separate irreversibly, leading to incorrect delivery of lookup messages. In this thesis, I prove that the pure-join model of Pastry is correct w.r.t. delivery of lookup messages. I also show that in the pure-join model, correctness still holds using a node join process that is simpler than the ones proposed for Pastry, for example in [21, 27].

In what follows, I discuss some of the relevant literature on the subject of formal verification of P2P protocols. Next, I describe the contribution of this thesis to this research area. This is followed by a brief description of the tools and techniques used in this thesis, in particular the proof assistant TLA$^+$. Finally, I give a short outline of the remainder of this thesis.

Related Work

Chord [38] has been one of the most studied DHT implementations in recent years, and operates in a way that is very similar to Pastry. It implements a distributed hash table as follows. A consistent hash function assigns each key and each live Chord node an m-bit identifier (a node's identifier is determined by hashing its IP address, while a key's identifier is determined by hashing the key). The identifier space is arranged as a "circle" called the Chord ring, containing the 2^m possible identifiers ranging from 0 to $2^m - 1$. A key k is assigned to the first node n whose identifier is equal to or follows that of k on the ring; n is called the *successor* of k. Each node n maintains a *successor list* of its successor nodes on the ring, and a *finger table* for efficient routing, which contains the addresses of up to m nodes at different "distance classes" from n. In particular, the ith entry is the first node at least 2^{i-1} identifiers after n on the ring. Chord nodes employ a periodic "stabilization algorithm" to keep their successor lists and finger tables up to date. Correct lookups can be guaranteed if each node knows its immediate successor on the ring. Due to node failure, however, this may not be the case. The effect of node failure is mitigated by replication, hence the successor lists; the bigger the size r of the successor list, the higher the probability that a lookup request for a key returns the correct node.

Bakhshi et al. describes an abstract model for structured P2P networks with a ring topology in π-calculus, and uses this model to verify the stabilization algorithm of Chord by establishing weak bisimulation between the specification of Chord as a ring network and the implementation of the stabilization algorithm [4]. This is a pure-join model in which node failure is not taken into account, and features such as finger tables and successor lists are not modeled.

Zave has done a lot of work verifying Chord. Using Alloy to formally model and verify Chord, she shows that the pure-join protocol—i.e. where nodes do not leave the network—is correct, but that the full version of the protocol may not maintain the claimed invariants [44]. In [45], she presents a full version of Chord (where both node joins and leaves are modeled) with a partially-automated proof of correctness. The correctness of this version of Chord relies on the assumption that there is a *stable base* of $r+1$ permanent ring members, where r is the size of the successor list.

$\mathcal{DKS}$ [1] is a distributed hash table implementation, where the name is short for *distributed k-ary search*. $\mathcal{DKS}(N, k, f)$ describes a P2P protocol for a network of maximum size N, a search arity of k within the network, and a fault-tolerance parameter f. Briefly, distributed k-ary search performs a lookup for any key t in at most $\log_k(N)$ hops. Initially, the search space is the entire key space of the hash table. At each step of the distributed search, the current search space is divided into k equal parts, each part being the responsibility of a different node. This partitioning is repeated until each partition contains only one element, at which point the part containing key t is the lookup target.

$\mathcal{DKS}$ can be seen as a generalization of other systems like Chord, Pastry or Tapestry. Like Chord, $\mathcal{DKS}$ also has a ring structure, though, according to the

authors, this structure is arbitrary and the algorithms provided work for other network topologies as well. The distinguishing feature of $\mathcal{DKS}$ is that unlike other protocols, it does not employ an active correction routine for the nodes' routing tables or other local information. Instead, routing tables are corrected on-the-fly while performing lookups and insertions. This idea is based on the observation that, in P2P systems where the number of lookups and insertions is significantly higher than the number of node joins, leaves and failures, correcting routing tables on the fly during lookup and insertion operations saves the unnecessary bandwidth consumption incurred by a periodical correction mechanism.

$\mathcal{DKS}$ allows nodes to join and leave the network freely. Like Chord, a node joins a $\mathcal{DKS}$ network by sending a join request to its successor on the ring. Joins are "atomic" in the sense that a node i receiving more than one join request at a time has to handle at most one at a time. A node may leave gracefully by informing other nodes of its leaving, or fail abruptly. It is claimed that the system can still provide reliable lookup if no more than f consecutive nodes fail simultaneously.

The authors of $\mathcal{DKS}$ conduct some experiments using simulation to observe how lookup efficiency is affected by churn [1]. $\mathcal{DKS}$ is also described in detail in the Ph.D. thesis of Ghodsi [20]. In this thesis, Ghodsi discusses several issues such as concurrent joins and node failure, and claims that it is impossible to guarantee correctness where node failure is possible, due to the possibility of network separation. Another work on $\mathcal{DKS}$ is by Borgström et al., who use CCS for the formal verification of lookups in the static case of the protocol, i.e. without taking node joins or failure into account [9].

The most relevant work as well as the starting point of this thesis was done by Lu [29, 30, 27] on Pastry. Pastry has a ring structure that is very similar to Chord, and employs successor as well as predecessor lists for each node, called the "leaf set" of that node. Instead of Chord's finger table, each node possesses a routing table, whose entries are determined by the length of the common prefix of bits between the entry's ID and the ID of the node. Like Chord, correct delivery of lookup messages can be guaranteed in Pastry if each node has a correct leaf set, or at least correct information about its direct successor/predecessor on the ring, regardless the content of the routing table.

Lu models Pastry in TLA$^+$, and uses the accompanying TLC model checker and the TLAPS proof assistant to formally verify correct delivery of lookup (*get*) messages: *at any point in time, there is at most one node that answers a lookup request for a key, and this node must be the closest live node to that key.* As in the case of Chord, Lu discovers several problems in the original Pastry protocol. He also shows that the improvements proposed in later publications on Pastry, as in [21], still do not guarantee consistency. Finally, he presents a pure-join variant of Pastry, which he calls LuPastry, for which he verifies correct delivery under the strong assumption that nodes never fail. Notably, Lu's Pastry variant also enforces "atomic" joins where a live node may only facilitate the joining of one new node at a time, as in the case of $\mathcal{DKS}$. Lu's proof reduces correct delivery to a set of around 50 claimed invariants, which

are proven with the help of TLAPS. As such, LuPastry represents a major effort in the area of computer-aided formal verification of distributed algorithms.

Still, Lu's proof relies on many unproven assumptions relating to arithmetic and to protocol-specific data structures. This is likely due to the sheer size of the proof and the lack of maturity of the TLA$^+$ proof assistant at the time. Upon examining Lu's proof, I discovered counterexamples to several of the underlying assumptions. While I was able to prove weaker variants of many assumptions, this was not possible for others. In fact, I was able to find a counterexample to one of Lu's claimed main invariants, for which the TLA$^+$ proof was only possible because of incorrect assumptions. This has led me to redesign the overall proof of correctness for LuPastry, and, in the process, improve the TLA$^+$ specification of the protocol in many ways.

While working on the new proof, I also realized that the node join process of the protocol can be made significantly simpler without impacting correctness. In particular, the final "lease exchange" step, a handshaking step between a new node and its neighbor nodes before it becomes an active participant, is unnecessary for correctness in the join-only scenario. In fact, this exchanging of leases was not part of the original Pastry protocol published in [37], but introduced by the authors in a later paper, among other improvements to the protocol [21]. While not explicitly stated, the likely reason behind introducing this additional step was improving the accuracy of the leaf sets and, consequently, the consistency of lookups in the protocol. As Lu shows, however, this lease exchange step does not guarantee lookup consistency; the full dynamic protocol where nodes join and leave freely violates correct delivery of lookup messages, even with the implementation of lease exchange. I observe, on the other hand, that the pure-join model is correct without this step. As a result of this observation, I have formalized a simpler variant of the LuPastry specification where the lease exchange step is omitted, and proven correct delivery of messages for this simpler variant.

Contribution of This Thesis

The contributions of this thesis are as follows.

1. **LuPastry$^+$.** An improved version of the TLA$^+$ specification of LuPastry [27]. Aside from fixing bugs, the specification is rewritten in a more modular way using new operators which greatly improve the level of automation in the correctness proof.

2. **LuPastry$^+$ Correctness Proof.** A complete, rigorous proof of correct delivery of lookup messages for LuPastry$^+$ in TLA$^+$.

3. **Simplified LuPastry$^+$.** A variant of LuPastry$^+$ with a simpler node join process in which the lease exchange step is omitted.

4. **Simplified LuPastry$^+$ Correctness Proof.** A complete, rigorous proof of correct delivery of lookup messages for Simplified LuPastry$^+$ in

TLA$^+$. This proof is adapted from the correctness proof for LuPastry$^+$, and represents a success story in proof reuse.

This thesis presents a unique contribution in the following ways. First, it is a study of two dynamic variants of Pastry that is not only limited to the static view of the network, but also allows for nodes to join the network concurrently. Second, the form of verification used here is full theorem proving, instead of less rigorous methods such as simulation or model checking. Third, the proof of correctness for Simplified LuPastry$^+$, which was adapted from the original proof for LuPastry$^+$, presents a unique success story in proof reuse, particularly for such a large proof of over 32,000 proof interactions.

Tools and Techniques

Formal Verification

Formal verification of a (hardware or software) system is the act of checking the correctness of the system—i.e. that the system operates as intended by satisfying a set of specific requirements/properties—using formal methods of mathematics.

There are different approaches to formal verification that vary in rigor, and accordingly in the time and effort required to apply them. Regardless the approach taken, one has to start with a *specification* of the system: a model of the system describing its desired behavior, written in the mathematical notation of choice, such as logic, automata, Petri nets, process algebra or set theory [7]. A specification should be as precise, unambiguous, concise and complete as possible. Two approaches of formal verification are relevant to this thesis: model checking and theorem proving.

Model checking. Model checking consists of checking that every possible state the system can reach from its initial state fulfills the desired properties. For finite-state systems, this approach is rigorous. It is also possible to apply model checking on infinite models where infinite sets of states can be effectively represented finitely. In systems with arbitrary size, model checking can be used to verify instances of the system that have a fixed size, which helps find bugs and gain confidence in the correctness of the full system specification, but does not serve as proof. That is, model checking is helpful in proving the presence, rather than the absence, of bugs [6]. Model checking has the advantage of being fully automatic and requiring little interaction from the user. However, a typical problem with model checking is "state explosion", since for many practical systems, the state space is either infinite or increases exponentially with the number of variable parameters of the system [3].

Theorem proving. Deductive verification is the process of generating a collection of mathematical *proof obligations*, the truth of which implies the correctness of the system, which are then *discharged* or proven by a dedicated

theorem prover. An *automated theorem prover* attempts to prove or disprove a given proof obligation in a completely automated manner, without interference from the user. Such provers can be general-purpose provers like Spass [41] or Zenon [8], or SMT (Satisfiability Modulo Theory) provers like CVC4 [5] or Z3 [16], which handle proof obligations relating to a certain theory such as the theory of arithmetic or set theory.[3] For large systems, an *interactive theorem prover*, also called a *proof assistant*, is a software tool that assists a user in writing large proofs, where the individual proof obligations are sent to a back-end automated theorem prover. Applying theorem proving to large systems can be a very lengthy and expensive process, and requires a very deep understanding of the system and many interactions with the proof assistant.

TLA$^+$

TLA$^+$ is a formal specification language for modeling and verifying distributed systems. In TLA$^+$, one can write a specification of the system, assert properties of the system, and write proofs for these properties.

The specification language is based on Temporal Logic of Action (TLA), a variant of linear-time temporal logic designed by Leslie Lamport in 1994, for specifying system behavior, and untyped Zermelo-Fränkel set theory for describing data structures. Systems are specified as state machines over a tuple of state variables by defining a state predicate *Init* and a transition predicate *Next* that constrain the possible initial states and the next-state relation. Transition predicates (also called *actions*) are first-order formulas that contain unprimed and primed state variables for denoting the values of the variables in the state before and after the transition.

Asserted properties of TLA$^+$ specifications are also expressed as logic formulas. These properties can be verified for finite instances of the specification using TLC [43], TLA$^+$'s explicit-state model checker. For instances of infinite or arbitrary size, the properties can be proven by writing proofs in the TLA$^+$ Proof System TLAPS [15], shown in Figure 2.1. TLAPS is based on a hierarchical proof language; the user writes a TLA$^+$ proof in the form of a hierarchy of *proof steps*. Each step is interpreted by the *proof manager*, which generates corresponding proof obligations and passes them to automatic back-end provers, including Zenon, Isabelle/TLA$^+$, and SMT solvers. Larger steps that cannot be proven directly by any of the back-end provers can be broken further into sub-steps. Because the language is untyped, part of the proof effort consists in proving a *typing invariant* that expresses the shapes of functions and operators.

TLA$^+$ and tools can be used from within the TLA$^+$ toolbox. The toolbox is an all-in-one IDE that allows users to write TLA$^+$ specifications, run the TLC model checker, and write and verify proofs using the proof manager. The TLA$^+$ specification and proof language are briefly explained in Chapter 3.

[3] More accurately, since both of these theories are very expressive and undecidable, SMT solvers typically focus on decidable fragments of these theories such as quantifier-free linear arithmetic over integers or reals.

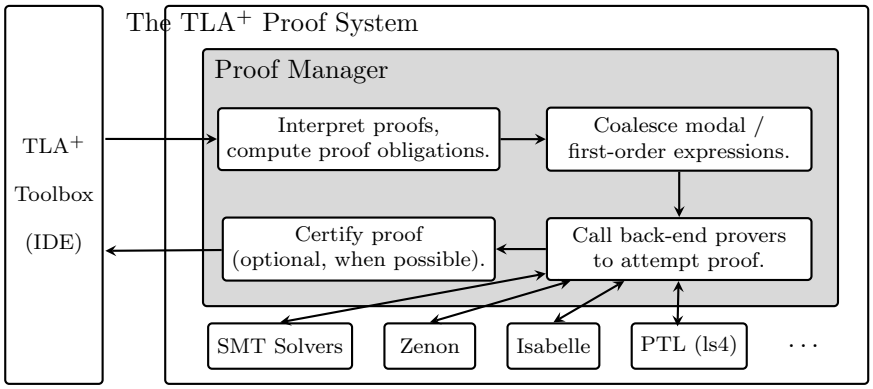


Figure 2.1 – The TLA$^+$ proof system.

Methodology

I prove the safety property "correct delivery" for two different variants of Pastry. The property states that *at any point in time, there is at most one node that answers a lookup request for a key, and this node must be the closest live node to that key* [27].

For each variant of Pastry, the methodology used for finding the proof is illustrated in Figure 2.2. The variant of Pastry in question is expressed as a TLA$^+$ specification, and the property correct delivery is proven by reducing it to a number of *inductive invariants* of the specification. A property P is an invariant of a given specification if it holds at every initial state of the specification, as well as all states reachable from the initial states. P is an inductive invariant if it holds at every initial state of the specification, and is preserved by all transitions. By this definition, every inductive invariant is an invariant, since properties valid at the initial state(s) and preserved by all transitions are valid at all reachable states.

The most challenging part of the proof process is coming up with the necessary and correct inductive invariants. The process of inferring the needed correctness invariants works as follows. First, I express a property P that seems both useful to the proof of correct delivery and also a correct inductive invariant of the specification. I use model checking to check if there are any counterexamples to this property in a small-size network of e.g. four or eight nodes. If TLC finds counterexamples, then P is not an invariant of the specification and needs to be reformulated accordingly. If no counterexamples are found, there is enough confidence to proceed to the next step of proving P to be an inductive invariant using TLAPS by induction. This process may still reveal counterexamples to P that could not be revealed by model checking, again requiring the property to be revised. If P can be proven as an invariant of the specification using TLAPS, it is added to the list of invariants that can be used to prove correct delivery. The proof of correct delivery itself is a

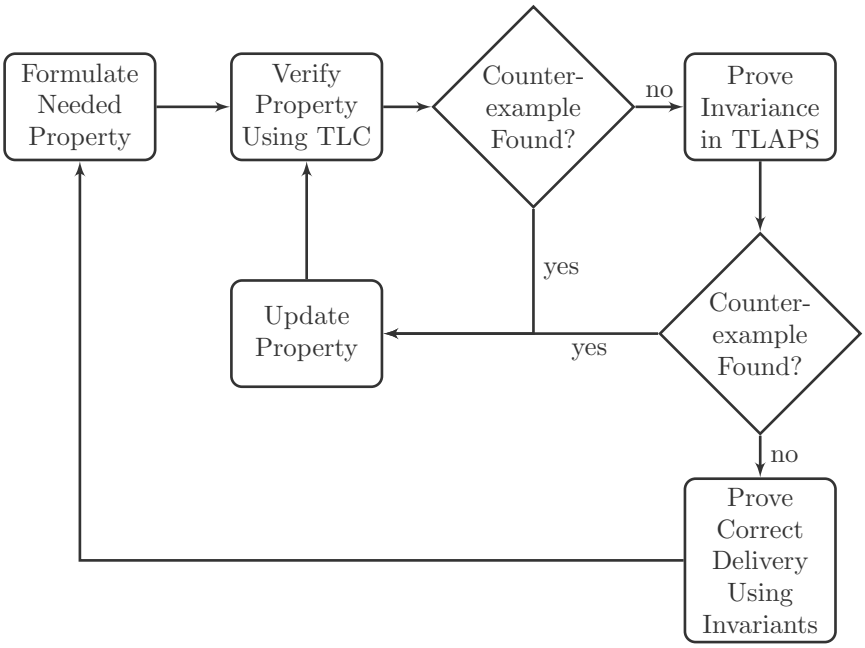


Figure 2.2 – Methodology used for the formal verification of Pastry using TLA$^+$.

process that runs in parallel to that of creating and modifying invariants; the invariants found are motivated by the proof of correct delivery, and, in turn, shape this proof.

Organization

This thesis is organized as follows. Preliminaries of peer-to-peer networks, distributed hash tables, and the TLA$^+$ language are presented in Chapter 3. In Chapter 4, I present LuPastry$^+$ and its specification in TLA$^+$. In Chapter 5, I describe the improvements that my new specification LuPastry$^+$ introduces to Lu's original specification of LuPastry. I also summarize my analysis of Lu's partial proof of correctness and describe counterexamples I have discovered to some of the assumptions and one claimed invariant in the proof. Finally, I show the structure of my new, complete correctness proof for LuPastry$^+$. The proof itself is described in Chapter 6. In Chapter 7, I present Simplified LuPastry$^+$, in which the lease exchange phase of the node join process is eliminated, and show how I adapt the TLA$^+$ proof for LuPastry$^+$ into a new proof of correctness for the simplified protocol. Finally, Chapter 8 contains concluding remarks and a discussion of limitations as well as possible future work.

Chapter 3

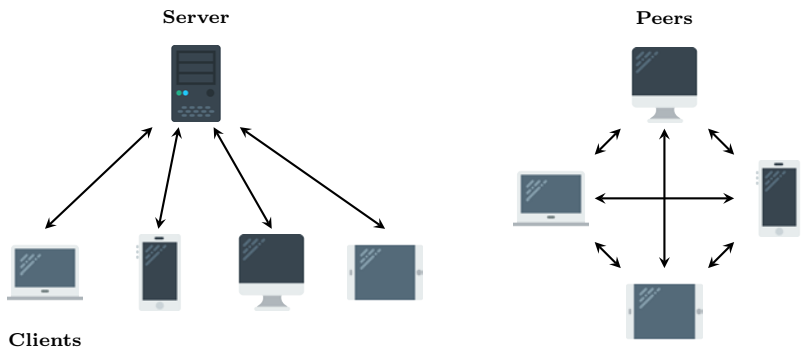
Preliminaries

3.1 Peer-to-Peer (P2P) Networks

In the traditional client/server architecture, which has been the de facto model for internet applications since the early days, a *client* makes a service request to a designated *server*, whose task is to receive and fulfill the request. This is a centralized network model where clients and servers are two distinct types of participants, with different capabilities and responsibilities, as shown in Figure 3.1a; clients are recipients of the service and do not share their own resources with other network participants, and servers are providers of the service and share their resources with the client. A typical example of the client/server architecture is the HyperText Transfer Protocol (HTTP), where the client initiates a request to retrieve a certain web page, and the server responds to the request. In this model, the centralized server holds most of the resources and therefore represents the most important part of the system, as well as the bottleneck.

In contrast to this model, *peer-to-peer* (P2P) is a decentralized application architecture in which all participant nodes, or *peers*, are both suppliers and consumers of the given service. Peers are considered to have equal capabilities, and tasks or work loads are distributed among all peers. Each peer makes a portion of its resources directly available to all other peers. Peers communicate with each other without the need for coordination by a central server (see Figure 3.1b). P2P came into popularity through Napster, the file sharing system released in 1999. Since then, it has become the underlying communications model for many internet applications in many domains such as content delivery, file sharing and multimedia streaming.

Participating nodes in a P2P network form a virtual *overlay network* on top of the physical network. Communication still happens directly over the underlying TCP/IP network, but at the application layer peers communicate directly with each other via logical links. Each "logical" path in the overlay network corresponds to an actual path in the underlying physical network. These logical overlay links make the P2P network independent from the topology of



(a) Client/Server Model (b) Peer-to-Peer Model

Figure 3.1 – The client/server model versus the decentralized P2P model.

the underlying physical network.

Because they utilize the resources of all peers, P2P networks have the characteristics of decentralized control, self-organization, adaptation and scalability.

However, all P2P networks were not created equal. P2P models can be classified as either *pure* or *hybrid*. Pure P2P networks are completely decentralized and server-free, and all nodes are seen as identical in responsibilities and capabilities. Pure P2P models can be further classified as either *structured* or *unstructured*. In their early days, pure P2P networks were *unstructured*. In an unstructured P2P network, there is no structure or topology that is globally imposed upon the different peers. Because of this lack of network topology, added to the lack of a central server, when a peer issues a search for certain piece of data in the network, the request is "flooded" through the network to find as many peers as possible that have the required data [19]. While easy to implement, this lack of structure causes a very high amount of unnecessary network traffic and CPU/memory usage [47, 19].

This lack of structure caused the trend to shift for a period of time towards a "hybrid" P2P architecture, which is a compromise between the client/server model and the pure P2P model. Hybrid networks are P2P networks that still employ some special servers (sometimes referred to as *supernodes*) for certain functions, for example to facilitate log-ins (joins), to coordinate part of the node-to-node communication, or to maintain network connectivity. Compared to pure unstructured P2P networks, hybrid models have the advantage in applications where certain functions like search benefit from a centralized structure while other functions benefit from the decentralized aggregation of the resources of all nodes [42].

A key problem with pure P2P networks, and in particular with large-scale networks, is how to efficiently manage the available resources. To compete with the hybrid model, a structure is needed to efficiently utilize the resources of

all peers. This motivated the rise of what is called structured P2P networks. In a *structured* P2P network, a specific topology is enforced, and the protocol ensures that using this structure, any node can efficiently search the network for a particular piece of data and retrieve it. The most common structure for a structured P2P network is a *distributed hash table*. Pastry, the focus of this thesis, is a structured P2P protocol that implements a distributed hash table. Other examples include Chord, CAN, Tapestry, Kademlia and DKS, all mentioned in the previous chapter. Structured P2P networks like Chord and CAN have been shown to outperform unstructured P2P networks with respect to search functionality by several orders of magnitude [19].

Distributed Hash Tables (DHT)

A *distributed hash table* (DHT) is a decentralized distributed system that provides a lookup service similar to that of a hash table, but where different nodes in the system are responsible for storing different (key, value) pairs (see Figure 3.2). Like a classic hash table, a DHT offers two main functions: $put(k, v)$ stores the value v at key k, and $get(k)$ retrieves the value associated with key k. As with a regular hash table, any participating node should be able to efficiently retrieve the value associated with a given key. Most DHT implementations promise a lookup performance of $O(\log N)$, where N is the number of active nodes. Due to the lack of a central server with a global view of the network, nodes in a DHT must collaborate to decide on their respective key storage range, and to route *put* and *get* requests to the appropriate node.

Nodes forming a DHT are assigned unique identifiers. These identifiers determine the topology of the network; messages from one node to another are routed through a number of intermediate nodes, the route being determined by node identifiers. Also, nodes determine their respective key storage range based on their own identifiers.

Distributed hash tables have a variety of potential applications. For example, they can be used to implement a distributed file-sharing system, where, instead of all files being stored on one central server, different files can be stored at different network nodes depending on the file's hash key. In order to retrieve a certain file, a node sends a lookup/*get* request with the hash key of the file. The underlying DHT protocol ensures that the lookup message arrives at the node storing the file in question.

From the perspective of this thesis, where correct delivery of messages is concerned, *put* requests can be seen as *get* requests where the node receiving the request performs the additional operation of storing a given value at a given key that it manages. Therefore, only lookup/*get* requests are modeled in the DHT protocols presented here.

3.2 Pastry

Pastry was first introduced in [37] as a "completely decentralized, fault-resilient, scalable, and reliable" DHT with good locality properties. Each node in the

Figure 3.2 – An example distributed hash table with four nodes and 256 keys.

Pastry network is assigned a unique identifier, which ranges from 0 to $2^{128} - 1$. This identifier is assigned randomly when the node joins the network, and it is assumed that the generated identifiers are uniformly distributed across the identifier space. For example, a node's ID may be generated as a cryptographic hash of the node's ID address or its public key. Because of this uniform distribution, nodes with adjacent identifiers are likely to be in different geographical locations.

As a DHT, the main function of Pastry is to efficiently route a given message with a given key to the node whose ID is numerically closest to that key among all live Pastry nodes. The expected performance of Pastry is $O(\log N)$ routing steps, where N is the number of live Pastry nodes.

The distinguishing feature of Pastry as opposed to other protocols like Chord is that Pastry takes locality into account. It seeks to minimize not only the number of routing steps in the overlay network, but also the distance traveled by the message according to some proximity metric such as the number of IP routing hops. Two applications based on Pastry are the large-scale storage utility PAST [18] and the publish/subscribe platform Scribe [13].

Node State Each node maintains a *routing table*, a *neighborhood set*, and a *leaf set*. For the purpose of routing, node identifiers can be seen as numbers

in base 2^b. The routing table of a node contains entries at various distances away from the node, based on the digits in the node's identifier. In particular, the entry at row r and column c of node i's routing table should contain a node whose ID shares an r digit prefix with i's ID, and where the $(r+1)^{st}$ digit is c. This setup of the routing table achieves logarithmic-time routing in the number of nodes. A node's neighborhood set contains N nodes that are closest to the node in terms of the predefined proximity metric (representing, for example, the geographical location). This set is taken into account during routing to enhance locality. A node's leaf set, on the other hand, contains the $2L$ nodes with identifiers that are numerically closest to the node's identifier on both the left and right sides of the ring. A typical value for L is 2^b. The leaf set serves as the base case for routing, but its main function is to maintain the structure of the Pastry ring and improve fault-tolerance. The leaf set is the part of the Pastry node state that is most relevant to the correctness proof presented in this thesis.

Node Joins When a new node wishes to join the network, the Pastry algorithm assigns it a random identifier i. It is assumed that the node knows about some Pastry node j. Node i sends a join request to j, and j forwards the request through the network until it reaches the numerically closest node k to i. Each node along the route of the join request message updates its information to include i, and send i a message to "introduce itself", attaching to the message its own state (leaf set, routing table and neighborhood set). When node k receives i's join request, it responds to it and sends i its own state. Node i builds its own leaf set, neighborhood set and routing table from the information it receives from k and other nodes. Finally, i contacts all nodes in its leaf set, neighborhood set and routing table, sending them a copy of its state. At this point, i is ready to fully participate in the Pastry network and route and receive messages.

Node Leaves/Failure Pastry nodes may fail or leave the network silently, without giving warning to other nodes. A Pastry node is considered failed when its neighbors in the ID space cannot communicate with it. In the event of node failure, the neighboring nodes lazily update their leaf sets as follows. If node i fails, and nodes j and k are its immediate left and right neighbors on the ID ring, then j contacts k and uses k's right leaf set to repair its own right leaf set, and k uses j's left leaf set to repair its own left leaf set. This procedure guarantees that nodes eventually repair their leaf sets if no more than L nodes with adjacent identifiers fail at the same time.

Since its introduction, Pastry has been the subject of several publications. In [11], the authors of Pastry present a study of its locality properties. They also introduce some refinements to the node join and leave process, but these refinements are mainly concerned with how routing tables are updated while maintaining good locality properties. In [12], the authors propose a design sketch of bootstrapping a Pastry network. In [10], the authors study attacks

aimed at preventing correct message delivery in Pastry and similar structured P2P networks, and presents defenses to these attacks, describing techniques for secure node joining, routing table maintenance and message routing in the presence of malicious nodes. In [31], the authors discuss techniques to reduce overhead incurred by the stabilization mechanism, which nodes use to repair their state under continuous joins and leaves.

Lease Exchange

In a 2005 paper, the authors introduce a new leaf set stabilization protocol for Pastry nodes [21]. In a Pastry network, a node's responsibilities is mostly determined by the portion of the ID space that its ID is numerically closest to. Therefore, consistency in Pastry is determined first and foremost by the reliability of nodes' leaf sets. The authors introduce a new leaf set stabilization protocol that aims to guarantee that, with high probability, only one node considers itself responsible for a given key at any time, despite routing anomalies. Essentially, this is the same property of interest in this thesis.

Briefly, the new leaf set stabilization mechanism relies on the concept of *key ownership*, which I also denote in this thesis by *key coverage*, following the convention of Lu in [27]. A node may only accept lookup messages and join request messages for a key that it *owns* or *covers*. Ideally, a node should cover exactly the set of keys that its ID is numerically closest to among all Pastry nodes. The join process is modified so that at the end of the join process, a node must explicitly agree on its key ownership/coverage range with both its right and left neighbor nodes (according to its leaf set). Each neighbor releases part of its key ownership, and the new node claims it. This takes place by exchanging a series of messages, which are called *leases* in this thesis. While it is not explicitly stated in the paper exactly which inconsistent behavior this additional lease exchange step was designed to mitigate, it is claimed that using this new mechanism, lookup messages can be guaranteed to arrive at the correct node "with high probability". In his Ph.D. thesis, Lu shows that this lease exchange step does not guarantee correct delivery of lookups in all cases [27]. In this thesis, I show that correct delivery of lookups is guaranteed in the pure-join model of Pastry, even without lease exchange.

3.3 The TLA$^+$ Language

I will illustrate the TLA$^+$ specification and proof language first using a concise summary taken from [24], and then by aid of an example of a simple specification, taken from Leslie Lamport's "TLA$^+$ Hyperbook" [26] with some slight modifications. The aim here is not to formally describe the TLA$^+$ syntax and semantics, but to give an intuitive understanding of the language that is sufficient for understanding the TLA$^+$ specifications and proof fragments presented in the next chapters. For the interested reader, Merz presents a very clear and compact explanation of TLA$^+$ in [33]. An in-depth explanation can be found

in Lamport's 2002 textbook "Specifying Systems" [25], or the more up-to-date "TLA$^+$ Hyperbook" [26], available online as open-source and continuously being updated by Lamport.

A Summary of TLA$^+$

A TLA$^+$ specification consists of three main types of statements: declarations of constant and variable parameters, definitions of functions and operators, and assertions of assumptions and theorems.

Constant parameters are parameters whose values may differ from one instance of the system to another, but that have fixed values during the entire execution of one system instance. Variable parameters are parameters whose values may change during system execution. The state of a system is defined by the values of its variable parameters.

Definitions of operators and functions as well as the assertions in the specification are expressed as logical formulas. TLA$^+$ distinguishes between two types of formulas: *transition* (or *action*) formulas and *temporal* formulas. Transition formulas describe states and state transitions. Temporal formulas describe behaviors, i.e. infinite sequences of states.

TLA$^+$ specifications are organized in *modules*. A module may incorporate declarations, definitions, assumptions and theorems from other modules $M_1, \ldots, M_n$ using the keyword EXTENDS.

EXTENDS $M_1, \ldots, M_n$

Constants and variables are declared using the keywords CONSTANT and VARIABLE (alternatively CONSTANTS and VARIABLES).

CONSTANTS $C_1, \ldots, C_n$

VARIABLES $V_1, \ldots, V_m$

Properties can be asserted as *assumptions* using the keyword ASSUMPTION or the synonymous ASSUME, as *axioms* using the keyword AXIOM, or as *theorems* using the keyword THEOREM or the synonymous LEMMA and COROLLARY. Assumptions, axioms and theorems may be given optional identifiers by inserting $id \triangleq$ after the keyword.

ASSUMPTION P

THEOREM $id \triangleq P$

Functions and operators are defined as follows. $F(x_1, \ldots, x_n) \triangleq exp$ defines the operator F with arity n such that $F(e_1, \ldots, e_n)$ equals exp with every occurrence of x_i replaced by e_i. $f[x_1 \in S_1, \ldots, x_n \in S_n] \triangleq exp$ defines a function f with arity n and domain $S_1 \times \cdots \times S_n$, such that $f[x_1, \ldots, x_n] = exp$ for all $x_1 \in S_1, \ldots, x_n \in S_n$.

$$F(x_1, \ldots, x_n) \triangleq exp$$

$$f[x_1 \in S_1, \ldots, x_n \in S_n] \triangleq exp$$

Functions are allowed to have recursive definitions, so the symbol f may occur in the expression exp.

TLA$^+$ expressions mainly inherit their syntax and semantics from both Zermelo-Fränkel set theory and first-order logic. In the examples above, I use P to denote expressions of assumptions and theorems, and exp for expressions of function and operator definitions, implying a distinction between logical formulas and general TLA$^+$ expressions which may or may not have a boolean type. However, TLA$^+$ is based on purely untyped set theory that does not even make a distinction between formulas and terms that exists in first-order logic. In TLA$^+$, every value is a set. The following, for example, is a syntactically-correct statement of a lemma in TLA$^+$.

$$\textsc{lemma}\ \mathit{Strange} \triangleq 3$$

Attempting to prove this lemma in TLA$^+$, however, would fail using any of the available back-end provers, since basically it is asking to prove that $3 = \textsc{true}$, which is not provable (or disprovable) given TLA$^+$ semantics.

Figure 3.3 shows the main constant operators of TLA$^+$ that are relevant for this thesis, extracted from Lamport's compact summary of TLA$^+$ syntax in [24]. The first two parts of the table list operators from set theory and first-order logic.

The TLA$^+$ keyword CHOOSE is Hilbert's choice operator. The TLA$^+$ expression $\textsc{choose}\ x : P(x)$ (or $\textsc{choose}\ x \in S : P$) denotes some fixed but arbitrary element x (in the set S) for which the property P holds, if some such x exists. If there is no such x, as in $\textsc{choose}\ x \in Nat : x * 0 = 1$, the result of the CHOOSE expression is not specified.

The next part in the table describes syntax for functions, tuples and records. A *tuple* $e = \langle e_1, \ldots, e_n \rangle$ is a special kind of function whose domain is $\{1, \ldots, n\}$ for a natural number n, such that $e[i] = e_i$ for $i \in \{1, \ldots, n\}$. A *string* is a tuple of characters where the string "abc" is the same as $\langle \mathrm{a}, \mathrm{b}, \mathrm{c} \rangle$. A *record* is a function whose domain is a finite set of strings. For a record $e = [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$, $e.h_i$ denotes $e[h_i]$, the h_i component of the record e.

Finally, the last part of the table describes syntax for actions and temporal operators.

Sets	
$=, \neq$	Equality and negated equality.
$\in$, $\notin$	Set membership and its negation.
$\cup, \cap, \subseteq, \setminus$	Union, intersection, subset and set difference.
$\{e_1, \ldots, e_n\}$	The set of elements $e_1, \ldots, e_n$.
$\{x \in S : P\}$	The set of elements x in set S satisfying P.
SUBSET S	The power set of set S.
Logic	
$\wedge, \vee, \neg, \Rightarrow, \equiv$	Conjunction, disjunction, negation, implication and equivalence.
$\forall x : P$	For all x, P holds.
$\exists x : P$	For some x, P holds.
$\forall x \in S : P$	For all x in the set S, P holds.
$\exists x \in S : P$	For some x in the set S, P holds.
TRUE, FALSE	Boolean constants for truth $\top$ and falsehood $\bot$.
BOOLEAN	The set {TRUE, FALSE}.
CHOOSE $x : P(x)$	An x satisfying P.
CHOOSE $x \in S : P(x)$	An x in the set S satisfying P.
Functions, Tuples and Records	
$f[e]$	The application of function f to expression e.
DOMAIN f	The domain of function f.
$[X \to Y]$	The set of functions with domain X and range Y.
$[x \in X \mapsto e]$	A function f such that $f[x] = e$ for x in X.
$[f$ EXCEPT $![e_1] = e_2]$	A function $\hat{f}$ equal to f except $\hat{f}[e_1] = e_2$. An @ in e_2 equals $f[e_1]$.
$e.h$	The h-component of record e.
$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$	A record whose h_i component is e_i.
$[h_1 : S_1, \ldots, h_n : S_n]$	The set of all records with h_i component in S_i.
$e[i]$	The ith component of tuple e.
$\langle e_1, \ldots, e_n \rangle$	An n-tuple with i^{th} component e_i.
$S_1 \times \cdots \times S_n$	The set of all n-tuples with i^{th} component in S_i.
$d_1 \ldots d_n$ or $d_1 \ldots d_m.d_{m+1} \ldots d_n$	Integers and real numbers where d_i is a digit.
Actions and Temporal Operators	
e'	The value of e in the final state of an action.
UNCHANGED e	$e' = e$
$[A]_e$	$A \vee (e' = e)$
ENABLED A	An A action is possible.

Figure 3.3 – A summary of TLA$^+$ syntax.

TLA$^+$ By Example

Consider two jugs, a big jug and a smaller jug. Initially, the two jugs are empty. The following are the possible actions: (1) empty the small jug, (2) empty the big jug, (3) fill the small jug from the faucet, (4) fill the big jug from the faucet, (5) fill the small jug by pouring from the big jug, and (6) fill the big jug by pouring from the small jug. Figure 3.4 shows a TLA$^+$ specification of this system in one module named *TwoJugs*.

First, because arithmetic operators such as $+, -$ are not built-in operators in TLA$^+$, the module imports TLA$^+$'s standard *Naturals* module. TLA$^+$ provides standard modules for *Naturals*, *Integers* and *Reals* that define arithmetic operators like the following.

$a + b,\ a - b,\ a * b$	Addition, subtraction, multiplication.
$a < b,\ a \leq b,\ a > b,\ a \geq b$	Binary comparisons.
$a \mathrel{..} b$	$\{n \in \mathbb{N} : a \leq n \leq b\}$
a^b	Exponentiation.
$a\%b$	$a \mod b$, such that $0 \leq a\%b < b$
$a \div b$	Division such that $a = b * (a \div b) + (a\%b)$

These modules also define the sets *Nat*, *Int* and *Real* of naturals, integers and real numbers respectively.

The next two lines declare the constant and variable parameters of the specification. The constants *BigCapacity* and *SmallCapacity* denote the maximum capacities of the big and small jug, respectively. The variables *big* and *small* denote the amount of water in the big and small jugs at a specific point in time, respectively.

The first definition of the module defines an operator *Min*, where $Min(x, y)$ returns the minimum of two numbers x and y. This operator will be useful for later definitions in the specification.

The next line defines *vars* to be the tuple of variables of the specification, $vars \triangleq \langle big, small \rangle$. At any point in time, the state of the system is defined by the values of the variables *vars*. Specifications are defined as state-transition systems, which means we need to define the *initial state*, and the *next-state relation*.

The initial state is described by a TLA$^+$ formula *Init*, which states that initially, $big = 0$ and $small = 0$. The next six definitions in the module specify the six possible actions of the system. In these formulas, unprimed occurrences of a variable name denote the value of this variable before the transition, and primed occurrences denote its value after the transition. The action *FillSmall*, for example, sets the next value of *small* to the maximum capacity of the small jug, keeping the other variable *big* unchanged. TLA$^+$ allows local declarations in the form of LET-IN statements, as can be seen in the definition of action *SmallToBig*.

The formula *Next* describes the next-state relation as a disjunction of all the possible actions. Note that disjunctions and conjunctions can be written in the form of "bulleted lists", where the indentation indicates the nesting level.

As mentioned above, TLA$^+$ distinguishes between two types of formulas: transition formulas and temporal formulas. Transition formulas describe states and state transitions, like the definitions of *Init*, *Min*, *Next* and the six action formulas in our example. Temporal formulas describe behaviors, i.e. infinite sequences of states. The system specification in our example is defined by the following temporal formula, using the box symbol as in classic temporal logic to mean "always".

$$Spec \triangleq Init \wedge \Box[Next]_{vars}$$

The formula *Spec* is a conjunction of the initial state *Init* and the formula $\Box[Next]_{vars}$, which specifies that every transition either satisfies the action formula *Next* or leaves *vars* unchanged.[1]

Proving in TLA$^+$

Let us complete the module in Figure 3.4 with more definitions and a meaningful proof, shown in Figure 3.5. Properties to be verified are also specified in TLA$^+$ as logical formulas. The invariant we would like to prove is the *type invariant* of the specification variables.

$$TypeInvariant \triangleq big \in 0 \,..\, BigCapacity \wedge small \in 0 \,..\, SmallCapacity$$

Since practically nothing can be proven about constant parameters (there are no definitions), we must make *assumptions* about their types, as in the assumption *ConstantTypes*. Operators and functions have TLA$^+$ definitions from which the types can be inferred, as in the lemma *MinType* which states the type of the operator *Min* by expanding its definition (using the keyword DEF).

We prove the type invariant by induction. The lemma *InitTypeInvariant* represents the induction base case, and proves that the type invariant holds initially by definition of the initial state and the type invariant itself. The lemma *NextTypeInvariant* represents the induction step and states that whenever the type invariant holds immediately before a transition in *Next*, it will hold immediately after the execution of that transition. The proof of this lemma is more involved, and proceeds by a case distinction on the possible actions in *Next* (part of the proof has been omitted for compactness). The proof consists of steps, each of which is processed by the proof manager into one or more *proof obligations* and sent to one or more of the back-end automated theorem provers to solve. It is possible to specify exactly which back-end prover to use on a particular step, as in the last step in the proof of this lemma which is sent to the SMT back-end. A step that is too big for the back-end prover to handle can be broken further into sub-steps; i.e. the proof is nested. Because this is

[1] Of course, the names *Spec*, *Init* and *Next* are not significant; they can be substituted for other names.

```
──────────────────────── MODULE TwoJugs ────────────────────────

EXTENDS Naturals

CONSTANTS BigCapacity, SmallCapacity
VARIABLES big, small

Min(x, y) ≜ IF x ≤ y THEN x ELSE y

vars ≜ ⟨big, small⟩

Init ≜ big = 0 ∧ small = 0

FillSmall ≜ small′ = SmallCapacity ∧ big′ = big

FillBig ≜ big′ = BigCapacity ∧ small′ = small

EmptySmall ≜ small′ = 0 ∧ big′ = big

EmptyBig ≜ big′ = 0 ∧ small′ = small

SmallToBig ≜
  LET poured ≜ Min(big + small, BigCapacity) − big
  IN  ∧ big′ = big + poured
      ∧ small′ = small − poured

BigToSmall ≜
  LET poured ≜ Min(big + small, SmallCapacity) − small
  IN  ∧ big′ = big + poured
      ∧ small′ = small − poured

Next ≜ ∨ FillSmall
       ∨ FillBig
       ∨ EmptySmall
       ∨ EmptyBig
       ∨ SmallToBig
       ∨ BigToSmall

Spec ≜ Init ∧ □[Next]_vars

──────────────────────────────────────────────────────────────
```

Figure 3.4 – TLA$^+$ specification of the “Two Jugs” problem.

$TypeInvariant \triangleq big \in 0 \,..\, BigCapacity \land small \in 0 \,..\, SmallCapacity$

ASSUME $ConstantTypes \triangleq$
$\quad \land\, BigCapacity \in Nat$
$\quad \land\, SmallCapacity \in Nat$
$\quad \land\, BigCapacity > SmallCapacity$

LEMMA $MinType \triangleq$
$\quad \forall\, x,\, y \in Nat : Min(x,\, y) \in Nat \land Min(x,\, y) \leq x \land Min(x,\, y) \leq y$
PROOF BY DEF Min

LEMMA $InitTypeInvariant \triangleq Init \Rightarrow TypeInvariant$
PROOF BY $ConstantTypes$ DEF $Init$, $TypeInvariant$

LEMMA $NextTypeInvariant \triangleq$
$\quad TypeInvariant \land [Next]_{vars} \Rightarrow TypeInvariant'$
PROOF
$\langle 1 \rangle$ SUFFICES ASSUME $TypeInvariant$, $[Next]_{vars}$ PROVE $TypeInvariant'$
OBVIOUS
$\langle 1 \rangle 1$.CASE $FillSmall$
$\quad$ BY $\langle 1 \rangle 1$ DEF $FillSmall$, $TypeInvariant$
$\langle 1 \rangle 2$.CASE $FillBig$
$\quad$ BY $\langle 1 \rangle 2$ DEF $FillBig$, $TypeInvariant$
$\langle 1 \rangle 3$.CASE $EmptySmall$
$\quad$ BY $\langle 1 \rangle 3$, $ConstantTypes$ DEF $EmptySmall$, $TypeInvariant$
$\langle 1 \rangle 4$.CASE $EmptyBig$
$\quad$ BY $\langle 1 \rangle 4$, $ConstantTypes$ DEF $EmptyBig$, $TypeInvariant$
$\langle 1 \rangle 5$.CASE $SmallToBig$
$\quad$ PROOF OMITTED
$\langle 1 \rangle 6$.CASE $BigToSmall$
$\quad$ PROOF OMITTED
$\langle 1 \rangle 7$.CASE UNCHANGED $vars$
$\quad$ BY $\langle 1 \rangle 7$ DEF $TypeInvariant$, $vars$
$\langle 1 \rangle$ QED BY SMT, $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$, $\langle 1 \rangle 4$, $\langle 1 \rangle 5$, $\langle 1 \rangle 6$, $\langle 1 \rangle 7$
$\qquad\qquad$ DEF $Next$, $vars$

THEOREM $AlwaysTypeInvariant \triangleq Spec \Rightarrow \Box TypeInvariant$
BY PTL, $InitTypeInvariant$, $NextTypeInvariant$ DEF $Spec$

Figure 3.5 – TLA$^+$ example proof for the "Two Jugs" problem.

a simple example, all proof steps have a depth of 1, as indicated by the prefix $\langle 1 \rangle$.

Finally, we prove the invariance of *TypeInvariant* in the specification *Spec* using the two previous lemmas. This last theorem is proven using the Propositional Temporal Logic prover LS4.

Chapter 4

LuPastry$^+$

This chapter presents LuPastry$^+$ and its specification in TLA$^+$. LuPastry$^+$ is based on Lu's specification of Pastry, LuPastry [27]. Essentially, LuPastry$^+$ is an implementation of LuPastry with improved TLA$^+$ definitions of functions and operators and a number of bug fixes. As will be discussed in Chapter 5, the improvements introduced in LuPastry$^+$ are not only necessary for a rigorous proof (because of the bug fixes), but also result in a shorter, more modular and more readable proof of correctness.

4.1 An Overview of the LuPastry$^+$ Model

The Network Ring, Coverage and Leaf Sets

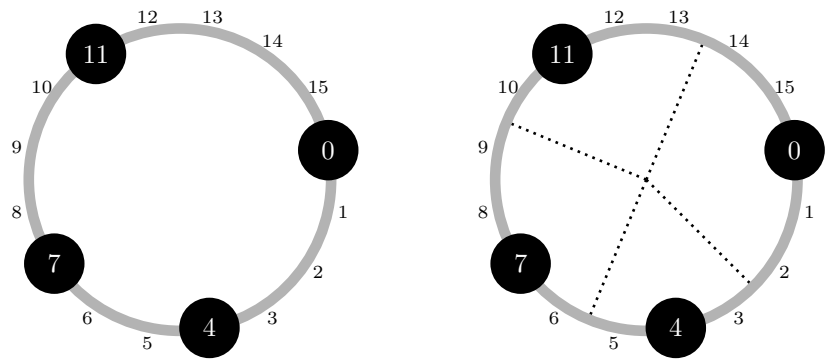
In Pastry, the set of possible *keys* is the interval $0 \mathinner{..} 2^M - 1$ for some positive integer M. Every live Pastry node is randomly assigned a unique *identifier* from the same key space—i.e. node identifiers are also keys. The Pastry key space can therefore be visualized as a ring with size 2^M (see Figure 4.1a).

I use the convention that keys increase in a clockwise direction on the ring. I also use the notion of *right* and *left* sides of a key: 0 is directly on the *left* side of 1, while 2 is directly on its *right* side.

For the implementation of a distributed hash table, live nodes[1] need to distribute among them the responsibility for storing (key, value) pairs. Each live node needs to determine its *coverage*: a contiguous range of keys, including the node's own ID, that the node is responsible for, or *covers*. If a node i covers key k, then i considers itself (1) the proper recipient of all lookup messages addressed to k, and (2) the node responsible for facilitating the joining of any new node that is assigned ID k. In the absence of a central server and shared memory, live nodes need to rely on message passing and local information to agree on a proper division of coverage.

An ideal distribution of coverage among live nodes is shown in Figure 4.1b: each two consecutive live nodes on the ring divide the coverage of keys that lie

[1] Fully-active network members that are not still in the process of joining.



(a) A Pastry ring of size 16, with 4 live nodes.

(b) Ideal distribution of coverage among live nodes.

Figure 4.1 – An example of a Pastry ring of size 16, and the ideal distribution of coverage among its live nodes.

between them (roughly) equally. For example, consider node 11 in the figure. On one side of the node, there are four keys that lie between it and node 0, which are $\{12, 13, 14, 15\}$. Therefore, node 11 should cover the two keys closest to it, $\{12, 13\}$, while node 0 should cover the other two keys, $\{14, 15\}$. On the other side of node 11, there are three keys between itself and node 7. Because there is an odd number of keys, the tie is arbitrarily (but deterministically) broken in favor of the left side. Therefore, node 7 covers the two keys closest to it, namely $\{8, 9\}$, while node 11 covers the key 10. So the coverage of node 11 should be the keys $\{10, 11, 12, 13\}$.

A node i determines its coverage by maintaining a *leaf set*: a set containing what i believes to be its L live neighbor nodes on both the left and right sides, for a positive integer L. The left and right *neighbors* of a leaf set *ls* are the closest nodes to the leaf set owner among all leaf set members.

In Figure 4.1b, for example, assuming correct leaf sets and $L = 2$, the leaf set of node 11 would look like this.

Left	Node	Right
$\{4, 7\}$	11	$\{0, 4\}$

The left and right neighbors of node 11 on the ring are nodes 7 and 0, respectively. Note that the left and right parts of the leaf set may *overlap*, i.e. one node may be in both the right and left leaf sets. Also, in this example, the leaf set is *complete* or *full*, i.e. it contains the maximum number of members L on both sides.

Finally, the coverage end-points of a node are computed as the midpoints between the node and its leaf set neighbors, and so the coverage of node 11 in the example would be the interval $[10, 13]$.

Node 10233102				
Prefix \ Digit	**0**	**1**	**2**	**3**
0	-0-2212102	1	-2-2301203	-3-1203203
1	0	1-1-301233	1-2-230203	1-3-021022
2	10-0-31203	10-1-32102	2	10-3-23302
3	102-0-0230	102-1-1302	102-2-2302	3
4	1023-0-322	1023-1-000	1023-2-121	3
5	10233-0-01	1	10233-2-32	
6	0		102331-2-0	
7			2	

Figure 4.2 – An example routing table of a node with ID 10233102, where node IDs are interpreted as 8 digits in base $2^B = 4$.

The Routing Table

Theoretically speaking, one could rely completely on the leaf set for the routing of messages among nodes. In order for node i to send a message to some other node j, i would look for j in its leaf set. If i does not find j, it forwards the message to some other leaf set member k_1 on the way to j. Node k_1, in turn, becomes responsible for routing the message further on, either to j or some other node k_2 along the way, and so on. Because leaf sets only contain the closest few neighbors of a node, routing in this way would take linear time in the number of nodes. While sufficient for correct routing, this method is highly inefficient.

Instead, it is possible to achieve logarithmic-time routing if each node maintains a *routing table*, which contains not only the closest neighbors to it, but nodes at different "distance classes" away from it. For example, each node i may keep the addresses of nodes that are roughly at distances 2, 4, 8, 16, ... and so on away from i.

In Pastry, this is implemented as follows. The protocol takes as a parameter a positive number B. All keys on the ring are interpreted as numbers in base 2^B. Note that M has to be divisible by B. And so a key or node ID k will have $M \div B$ digits in base 2^B, each digit in the range $0 \,..\, 2^B - 1$.

The routing table of node i is essentially a matrix, with $M \div B$ rows and 2^B columns. The value at row r and column c in the matrix is either *NIL* if no entry is present, or a node j that has the same r-digit prefix as i, and whose $(r+1)^{st}$ digit is c.

Routing tables can be illustrated using the following example taken directly from [37]. In this example, $M = 16$, $B = 2$ and therefore the base is $2^B = 4$. A node ID is read as an 8-digit number in base 4. Shaded cells indicate the corresponding digit in the node's ID. Figure 4.2 shows an example routing table for a node with ID 10233102. Entries at row r have a shared prefix of exactly r digits with the node.

Node Failure

LuPastry$^+$ operates under the same assumption of LuPastry that nodes do not fail or leave the network.

Node Join

The node join process is illustrated in Figure 4.3. In LuPastry$^+$, each node is either "Dead" (not shown), "Waiting" (white), "OK" (gray) or "Ready" (black). "Dead" nodes simply refer to keys in the key space that are not assigned as identifiers to some node on the network. Only "Ready" nodes facilitate the joining of new nodes into the network, and based on the join model in the original LuPastry, a "Ready" node may only help at most one new node join the network at a time.

Suppose some "Dead" node i that decides to join the network. It is assumed that node i has the means of obtaining the address of some "Ready" node j, for example through some dedicated server. Node i starts the join process by changing its status to "Waiting" and sending a *join request* to node j. Node j forwards the join request to the "Ready" node k that covers key i. We denote node k by the node *responsible* for i.

When node k receives the join request message, and when it is not handling the joining of any other node, it responds to i with a *join reply* message. Node k attaches its own leaf set and routing table to the join reply message, so that node i can use them to eventually build its own leaf set and routing table.

When node i receives k's join reply message, containing k's leaf set, it begins the *probing* phase: in order for node i to construct its proper leaf set (and routing table), i sends *probe* messages to the nodes in the leaf set received from k. More accurately, i only sends probe messages to the nodes that are close enough to it on the ring to be contained in its leaf set.

All non-dead nodes that receive the probe from i potentially add i to their leaf set, and respond to i with a *probe reply* message, attaching their own leaf set. Node i continues to receive probe replies and to probe more newly-discovered nodes. This process continues until i has probed all nodes it has heard of that are close enough to i to be in i's leaf set.

When i has received probe replies to all its probes, and no new nodes have been discovered, i finishes probing and changes its status to "OK". "OK" nodes can be viewed as semi-live nodes that have finished constructing their leaf sets and routing tables, but must await confirmation from their leaf set neighbors before becoming fully live, or "Ready". In order for i to become "Ready", i has to exchange *leases* with both its left and right leaf set neighbors. Note that one of those neighbors must be k, since k is the facilitator of i's join process and therefore must be closest to i on the ring on either the right or the left side (this is also proven in the correctness proof).

Node i sends out *lease request* messages to both its leaf set neighbors. If i's neighbor is "Ready" or "OK", and also considers i to be its neighbor, it grants i the lease in a *lease reply* message. When i has received lease replies

from both its neighbors, it switches to "Ready", and grants its neighbors leases in turn by sending them lease reply messages. When k receives i's lease reply message, it may help other new nodes join the ring.

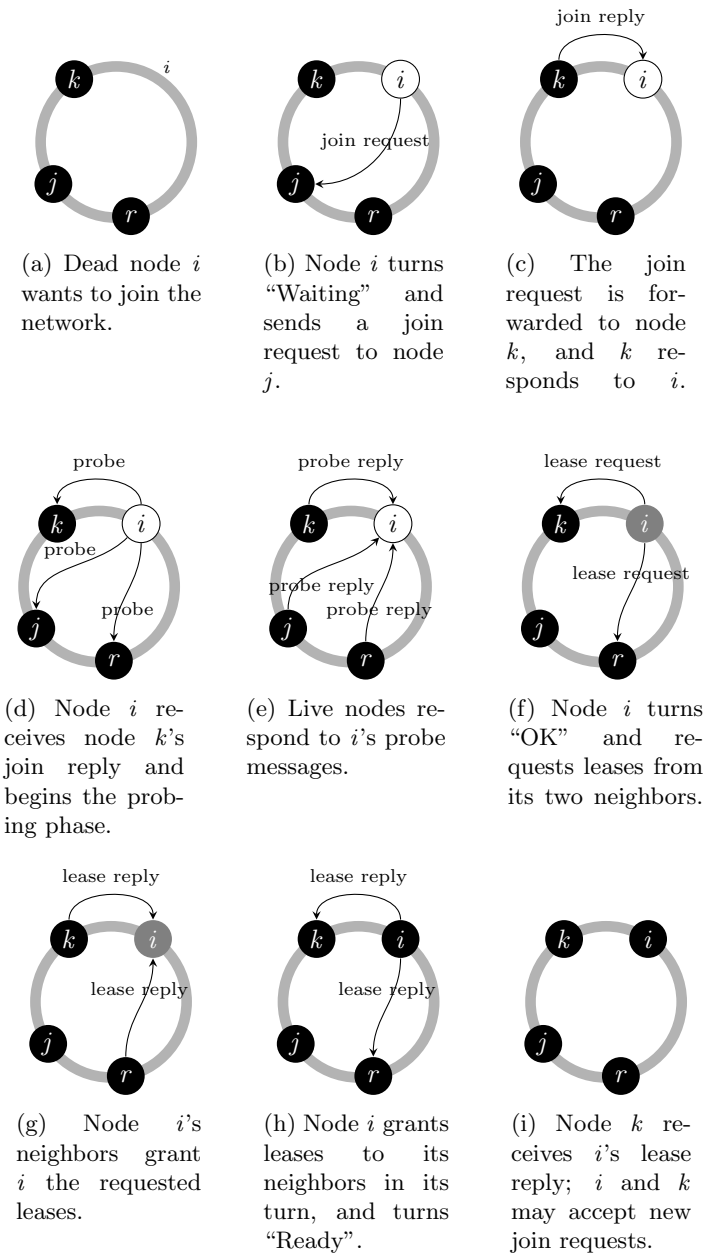
Because i is joining through k, k must be Ready, and i and k must be in agreement that they are each other's leaf set neighbors. Therefore, k must grant i's lease request. Let us assume k is i's left neighbor, and i is k's right neighbor, as in Figure 4.3. There is a possibility, however, that i's other neighbor r does not see i as its left neighbor. Provably, this can only happen if there is another Waiting node u between i and r that has started a join process through r, and has become r's left leaf set neighbor. In this case, i's lease request to r is denied, and i cannot turn "Ready"; it must autonomously repeat the lease request again at a later time until it is granted. This scenario is not relevant for the safety property this thesis is interested in verifying. However, under fairness conditions, i will eventually hear from u and update its leaf set to have u as its right leaf set neighbor. Node i will repeat its lease request, this time to u, and u will grant this request. Eventually, i will be able to turn "Ready".

Communication

Communication is modeled as a set of messages that are "in transmission", i.e. pending messages that have been sent from the source node, but not yet received by the destination node. Nodes send messages by adding them to this set, and receive messages by removing them from this set. There is no guarantee as to the order in which sent messages are read by the destination node; the messages are not timestamped and message interleaving is fully allowed. However, as will be seen in the TLA$^+$ specification, a node may be prevented from "receiving" a certain message—i.e. removing it from the set of pending messages—until certain preconditions have been fulfilled. For example, a Waiting node may not pick up any probes addressed to it until its leaf set is non-empty. Message loss is simulated by arbitrarily dropping messages from the set of pending messages.

4.2 The TLA$^+$ Specification of LuPastry$^+$

The complete picture of the TLA$^+$ specification of LuPastry$^+$ is shown in Figure 4.4. At the very bottom, the specification starts with the declaration of constant parameters. Then, the network ring and the accompanying distance functions are defined, followed by definitions of the necessary data structures or "types", like the leaf set, routing table and messages exchanged between nodes. The dynamic aspect of the specification is described as a state-transition system, where a state is defined by the values of the variables of the specification. The variables are declared, followed by the definition of the initial state, and a next-state relation given by a set of possible *actions* or *events*.



(a) Dead node i wants to join the network.

(b) Node i turns "Waiting" and sends a join request to node j.

(c) The join request is forwarded to node k, and k responds to i.

(d) Node i receives node k's join reply and begins the probing phase.

(e) Live nodes respond to i's probe messages.

(f) Node i turns "OK" and requests leases from its two neighbors.

(g) Node i's neighbors grant i the requested leases.

(h) Node i grants leases to its neighbors in its turn, and turns "Ready".

(i) Node k receives i's lease reply; i and k may accept new join requests.

Figure 4.3 – The LuPastry/LuPastry$^+$ join protocol.

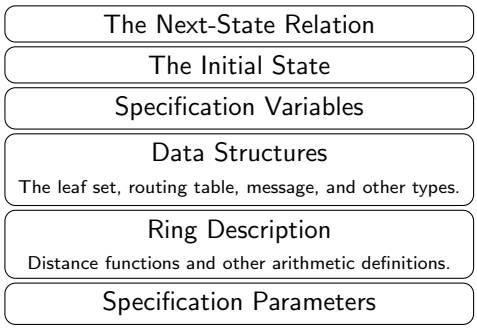


Figure 4.4 – An overview of the LuPastry$^+$ specification in TLA$^+$.

Constant Parameters

The LuPastry$^+$ specification takes the following constant parameters.

Definition 1 (LuPastry$^+$ Constants)**.**

CONSTANTS A, B, M, L, NIL

These constants can be described as follows.

A	The set of nodes that are live in the initial state.
B	Keys are interpreted as numbers in base 2^B.
M	Decides the size of the Pastry ring 2^M. Keys and node identifiers are in the range $0 \mathrel{..} 2^M - 1$.
L	The size of the left (right) leaf set of a node.
NIL	Represents an empty slot in the routing table.

Ring Description and Distance Metrics

The ID interval and ring size are defined as follows.[2]

Definition 2 (Ring Interval and Size)**.**

$$RingSize \triangleq 2^M$$
$$I \triangleq 0 \mathrel{..} (RingSize - 1)$$

Two main distance metrics are used in LuPastry$^+$, the *clockwise* distance from one node to another, and *absolute* (shortest) distance between two nodes.

[2] As will be discussed in Chapter 5, the actual LuPastry$^+$ definitions of ring size, leaf set operations, leaf set neighbors and coverage rely on auxiliary operators but are "flattened" here for clarity.

Definition 3 (Clockwise Distance).

$$
\begin{aligned}
&\mathit{ClockwiseDistance}(x,\ y) \triangleq \\
&\quad \text{IF } y \geq x \text{ THEN } y - x \text{ ELSE } \mathit{RingSize} - x + y
\end{aligned}
$$

Definition 4 (Absolute/Shortest Distance).

$$
\begin{aligned}
&\mathit{AbsoluteDistance}(x,\ y) \triangleq \\
&\quad \text{LET } d1 \triangleq \mathit{ClockwiseDistance}(x,\ y) \\
&\qquad\ \ d2 \triangleq \mathit{ClockwiseDistance}(y,\ x) \\
&\quad \text{IN } \ \text{IF } d1 \leq d2 \text{ THEN } d1 \text{ ELSE } d2
\end{aligned}
$$

In Figure 4.1, for example, where $\mathit{RingSize} = 16$, the clockwise distance from node 1 to node 14 is $\mathit{ClockwiseDistance}(1, 14) = 13$, but the shortest distance between them is counter-clockwise, $\mathit{AbsoluteDistance}(1, 14) = 3$.

Leaf Sets and Routing Tables

The leaf set data structure is defined as the following set of records.

Definition 5 (Leaf Set).

$$
\begin{aligned}
&\mathit{LeafSet} \triangleq \{ls \in [\mathit{node} : I,\ \mathit{left} : \text{SUBSET } I,\ \mathit{right} : \text{SUBSET } I] : \\
&\quad \wedge\ ls.\mathit{node} \notin ls.\mathit{left} \\
&\quad \wedge\ ls.\mathit{node} \notin ls.\mathit{right} \\
&\quad \wedge\ \mathit{Cardinality}(ls.\mathit{left}) \leq L \\
&\quad \wedge\ \mathit{Cardinality}(ls.\mathit{right}) \leq L\}
\end{aligned}
$$

The *content* of a leaf set ls is the set of all nodes contained in this leaf set, including its owner $ls.\mathit{node}$. The *empty leaf set* of a node i is a leaf set whose owner is i and which has no members in the left or right sides.

Definition 6 (Leaf Set Content).

$$\mathit{LeafSetContent}(ls) \triangleq ls.\mathit{left} \cup ls.\mathit{right} \cup \{ls.\mathit{node}\}$$

Definition 7 (Empty Leaf Set).

$$\mathit{EmptyLS}(i) \triangleq [\mathit{node} \mapsto i,\ \mathit{left} \mapsto \{\},\ \mathit{right} \mapsto \{\}]$$

A new leaf set may be obtained by adding nodes to, or removing nodes from, another leaf set. The two operations are almost identical; the only difference is the pool of nodes from which the new leaf set members are selected. When adding a set of nodes a to a leaf set ls, the selection pool is the original members of ls and the new nodes in a. When removing a set of nodes a from a leaf set ls the selection pool is the original members of ls, excluding those in a. Members of the new leaf set are selected as the (right and left) closest nodes to the leaf set owner $ls.node$.

Definition 8 (Adding To Leaf Set)**.**

$$
\begin{array}{l}
AddToLS(a, ls) \triangleq \\
\quad \text{LET } i \triangleq ls.node \\
\qquad C \triangleq (ls.right \cup ls.left \cup a) \setminus \{i\} \\
\qquad NL \triangleq \text{IF } Cardinality(C) \leq L \text{ THEN } C \\
\qquad\qquad \text{ELSE CHOOSE } S \in \text{SUBSET } C : \\
\qquad\qquad\quad Cardinality(S) = L \wedge \forall\, x \in (C \setminus S),\ y \in S : \\
\qquad\qquad\qquad ClockwiseDistance(y, i) < ClockwiseDistance(x, i) \\
\qquad NR \triangleq \text{IF } Cardinality(C) \leq L \text{ THEN } C \\
\qquad\qquad \text{ELSE CHOOSE } S \in \text{SUBSET } C : \\
\qquad\qquad\quad Cardinality(S) = L \wedge \forall\, x \in (C \setminus S),\ y \in S : \\
\qquad\qquad\qquad ClockwiseDistance(i, y) < ClockwiseDistance(i, x) \\
\quad \text{IN } [node \mapsto i,\ left \mapsto NL,\ right \mapsto NR]
\end{array}
$$

Definition 9 (Removing From Leaf Set)**.**

$$
\begin{array}{l}
RemoveFromLS(a, ls) \triangleq \\
\quad \text{LET } i \triangleq ls.node \\
\qquad C \triangleq (ls.right \cup ls.left) \setminus (\{i\} \cup a) \\
\qquad NL \triangleq \text{IF } Cardinality(C) \leq L \text{ THEN } C \\
\qquad\qquad \text{ELSE CHOOSE } S \in \text{SUBSET } C : \\
\qquad\qquad\quad Cardinality(S) = L \wedge \forall\, x \in (C \setminus S),\ y \in S : \\
\qquad\qquad\qquad ClockwiseDistance(y, i) < ClockwiseDistance(x, i) \\
\qquad NR \triangleq \text{IF } Cardinality(C) \leq L \text{ THEN } C \\
\qquad\qquad \text{ELSE CHOOSE } S \in \text{SUBSET } C : \\
\qquad\qquad\quad Cardinality(S) = L \wedge \forall\, x \in (C \setminus S),\ y \in S : \\
\qquad\qquad\qquad ClockwiseDistance(i, y) < ClockwiseDistance(i, x) \\
\quad \text{IN } [node \mapsto i,\ left \mapsto NL,\ right \mapsto NR]
\end{array}
$$

Leaf set neighbors of a leaf set ls are defined to be the closest nodes to the leaf set owner $ls.node$ among all leaf set members.

Definition 10 (Leaf Set Neighbors)**.**

$$
\begin{array}{l}
LeftNeighbor(ls) \triangleq \\
\quad \text{IF } ls.left = \{\} \text{ THEN } ls.node \\
\quad \text{ELSE CHOOSE } n \in ls.left : \forall\, m \in ls.left : \\
\qquad ClockwiseDistance(n,\, ls.node) \leq ClockwiseDistance(m,\, ls.node) \\
\\
RightNeighbor(ls) \triangleq \\
\quad \text{IF } ls.right = \{\} \text{ THEN } ls.node \\
\quad \text{ELSE CHOOSE } n \in ls.right : \forall\, m \in ls.right : \\
\qquad ClockwiseDistance(ls.node,\, n) \leq ClockwiseDistance(ls.node,\, m)
\end{array}
$$

A leaf set is *complete* if both the left and right parts contain L nodes.

Definition 11 (Complete Leaf Set)**.**

$$IsComplete(ls) \triangleq Cardinality(ls.right) = L \wedge Cardinality(ls.left) = L$$

Finally, the interval $[LeftCoverage(ls), RightCoverage(ls)]$ is the coverage region of leaf set ls. The TLA$^+$ predicate $Covers(ls, k)$ is true if key k is in the coverage range of leaf set ls. If every node knows its true right and left neighbors on the ring, this interval computes the ideal coverage as shown in Figure 4.1b. However, if a node's leaf set is not up-to-date, the leaf set neighbor of a node may not be its true live neighbor on the ring. Therefore, a node may under- or over-estimate its coverage region.

Definition 12 (Coverage)**.**

$$
\begin{array}{l}
LeftCoverage(ls) \triangleq \\
\quad \text{IF } LeftNeighbor(ls) = ls.node \text{ THEN } ls.node \\
\quad \text{ELSE } (LeftNeighbor(ls) + \\
\qquad (ClockwiseDistance(LeftNeighbor(ls),\, ls.node) \div 2 + 1))\%RingSize \\
\\
RightCoverage(ls) \triangleq \\
\quad \text{IF } RightNeighbor(ls) = ls.node \\
\quad \text{THEN } (RingSize + ls.node - 1)\%RingSize \\
\quad \text{ELSE } (ls.node + \\
\qquad ClockwiseDistance(ls.node,\, RightNeighbor(ls)) \div 2)\%RingSize \\
\\
Covers(ls,\, k) \triangleq \\
\quad ClockwiseDistance(LeftCoverage(ls),\, k) \\
\qquad \leq ClockwiseDistance(LeftCoverage(ls),\, RightCoverage(ls))
\end{array}
$$

Because the routing table is irrelevant to the correctness proof presented here, most TLA$^+$ specifics are omitted for the sake of compactness and clarity. Analogously to the leaf set, the routing table data structure is defined by *RoutingTable*. *EmptyRT* denotes the empty routing table (i.e. where all entries are *NIL*), and $RTContent(rt)$ is the set of nodes in routing table rt, excluding *NIL* entries. $AddToRT(a, rt, i)$ defines the operation of adding the set of nodes a to the routing table rt belonging to node i.

The TLA$^+$ formula $NextHop(i, j)$ determines the next node on the path from i to j, based on i's leaf set and routing tables. If there is no route possible, e.g. if i's leaf set and routing tables are empty, then $NextHop(i, j) = i$. If j is a member of i's leaf set or routing table, then $NextHop(i, j) = j$. Otherwise, $NextHop(i, j) = k$ for some $k \neq i, k \neq j$ such that k is closer to j than i is in terms of absolute distance on the ring.

Messages

Nodes can exchange messages of the following types: (1) a lookup message, "Lookup", (2) a routing error message, "IllegalRoute", (3) a join request, "JoinRequest", (4) a join reply, "JoinReply", (5) a probe message, "Probe", (6) a probe reply message, "ProbeReply", (7) a lease request message, "LeaseRequest", and (8) a lease reply message, "LeaseReply". A message in TLA$^+$ is defined as a record with a *destination* and a *content*. The content of a message contains all relevant information, such as the message type, its sender, or any other attachments like a leaf set or a routing table.

Definition 13 (Message).

$$
\begin{array}{lcl}
msg\_Lookup & \triangleq & [type : \{\text{"Lookup"}\},\ node : I] \\
msg\_NoLegalRoute & \triangleq & [type : \{\text{"NoLegalRoute"}\},\ key : I] \\
msg\_JoinRequest & \triangleq & [type : \{\text{"JoinRequest"}\},\ rt : RoutingTable, \\
 & & \quad node : I] \\
msg\_JoinReply & \triangleq & [type : \{\text{"JoinReply"}\},\ rt : RoutingTable, \\
 & & \quad ls : LeafSet] \\
msg\_Probe & \triangleq & [type : \{\text{"Probe"}\},\ node : I,\ ls : LeafSet, \\
 & & \quad failed : \text{SUBSET } I] \\
msg\_ProbeReply & \triangleq & [type : \{\text{"ProbeReply"}\},\ node : I,\ ls : LeafSet, \\
 & & \quad failed : \text{SUBSET } I] \\
msg\_LeaseRequest & \triangleq & [type : \{\text{"LeaseRequest"}\},\ node : I] \\
msg\_LeaseReply & \triangleq & [type : \{\text{"LeaseReply"}\},\ ls : LeafSet, \\
 & & \quad grant : \text{BOOLEAN}\ ] \\
MessageContent & \triangleq & msg\_Lookup \cup msg\_NoLegalRoute \\
 & & \quad \cup\ msg\_JoinRequest \cup msg\_JoinReply \\
 & & \quad \cup\ msg\_Probe \cup msg\_ProbeReply \\
 & & \quad \cup\ msg\_LeaseRequest \cup msg\_LeaseReply \\
Message & \triangleq & [destination : I,\ content : MessageContent]
\end{array}
$$

Variable Parameters

The state of a LuPastry$^+$ network is represented as the tuple *vars* of variables.

Definition 14 (LuPastry$^+$ Variables)**.**

$$vars \triangleq \langle MessagePool, Status, LeafSets, RoutingTables, Probing, Leases, Grants, ToJoin, Failed \rangle$$

MessagePool represents the set of messages currently in transmission. Messages are added to this set by the sending node and removed when they are received by the destination node. Variables *Status*, *LeafSets*, and *RoutingTables* are arrays whose i^{th} entries are the current status, leaf set, and routing table of node i. Similarly, $Probing[i]$ is the set of nodes that node i has probed but has not heard back from yet, $Leases[i]$ and $Grants[i]$ are the set of nodes i has acquired leases from, and granted leases to, respectively. Lastly, $ToJoin[i]$ designates the node that is currently joining through i, if any, otherwise $ToJoin[i] = i$. Lastly, $Failed[i]$ is the set of nodes that i suspects to have died. Because of the restriction imposed in LuPastry$^+$ that nodes do not leave the network, failure recovery is not modeled in the protocol and therefore the set $Failed[i]$ is always provably empty.

The Initial State

The initial state of the LuPastry$^+$ network is given by the formula *Init*.

Definition 15 (Initial State)**.**

$$\begin{aligned}
&Init \triangleq \\
&\quad \land\ MessagePool = \{\} \\
&\quad \land\ Status\ = [i \in I \mapsto \text{IF } i \in A \text{ THEN } \textit{"Ready"} \text{ ELSE } \textit{"Dead"}] \\
&\quad \land\ ToJoin = [i \in I \mapsto i] \\
&\quad \land\ Probing = [i \in I \mapsto \{\}] \\
&\quad \land\ Failed\ = [i \in I \mapsto \{\}] \\
&\quad \land\ Leases\ = [i \in I \mapsto \text{IF } i \in A \text{ THEN } A \text{ ELSE } \{i\}] \\
&\quad \land\ Grants = [i \in I \mapsto \text{IF } i \in A \text{ THEN } A \text{ ELSE } \{i\}] \\
&\quad \land\ LeafSets = [i \in I \mapsto \text{IF } i \in A \\
&\qquad\qquad\qquad\qquad \text{THEN } AddToLS(A, EmptyLS(i)) \\
&\qquad\qquad\qquad\qquad \text{ELSE } EmptyLS(i)] \\
&\quad \land\ RoutingTables = [i \in I \mapsto \text{IF } i \in A \\
&\qquad\qquad\qquad\qquad\qquad \text{THEN } AddToRT(A, EmptyRT, i) \\
&\qquad\qquad\qquad\qquad\qquad \text{ELSE } AddToRT(\{i\}, EmptyRT, i)]
\end{aligned}$$

Initially, there are no messages being exchanged, so *MessagePool* is empty. The status of each node i is Ready if i belongs to the set A of initially-live

nodes. No nodes are in the process of joining, so $ToJoin[i] = i$ for all i. No node is probing other nodes, and no nodes are suspected as failed. An A-node is assumed to have granted leases to, and obtained leases from, all other A-nodes. Also, A-nodes include each other in their leaf sets and routing tables, while Dead nodes have empty leaf sets and routing tables.

The Next-State Relation

The next-state relation defines the possible actions of the protocol and is given by the following TLA$^+$ formula *Next*.

Definition 16 (Next-State Relation)**.**

$$
\begin{aligned}
&Next \triangleq \\
&\quad \exists\, i, j \in I : \\
&\qquad \lor\ Lookup(i, j) \\
&\qquad \lor\ RouteLookup(i, j) \\
&\qquad \lor\ DeliverLookup(i, j) \\
&\qquad \lor\ Join(i, j) \\
&\qquad \lor\ RouteJoinRequest(i, j) \\
&\qquad \lor\ ReceiveJoinRequest(i) \\
&\qquad \lor\ ReceiveJoinReply(i) \\
&\qquad \lor\ ReceiveProbe(i) \\
&\qquad \lor\ ReceiveProbeReply(i) \\
&\qquad \lor\ RequestLease(i) \\
&\qquad \lor\ ReceiveLeaseRequest(i) \\
&\qquad \lor\ ReceiveLeaseReply(i) \\
&\qquad \lor\ LoseMessage
\end{aligned}
$$

The last action in the formula, *LoseMessage* models message loss by dropping an arbitrary message from the set *MessagePool*. This action is not relevant for proving the safety property of interest in this thesis, but is included in the specification since the underlying network is not assumed to be reliable. It can be seen from the definition of *Next* and the high-level description in Section 4.1 that, with the exception of action *LoseMessage*, the actions of LuPastry$^+$ can be divided into four main categories: lookups, join handling, probing and lease exchange. The next section describes these actions in more detail.

Actions In Detail

Lookups

Lookups are modeled using three actions. A lookup message for some key j is issued to any non-dead destination i through the action $Lookup(i, j)$.

A node i that receives a lookup message for some key j that it does not cover has to forward this message to the closest node k to key j that it can

find in its leaf set or routing table (possibly j itself). If no such k is found, an error message is returned. This is modeled in the action *RouteLookup*.

Of particular relevance to the correctness proof is the action *DeliverLookup*, where a node i receives a lookup request for a key j that it covers. Note that the actual response to the lookup message is irrelevant to correctness and therefore not modeled in the specification; the lookup message is simply received by i and discarded.

Definition 17 (Action Lookup)**.**

$$
\begin{array}{l}
Lookup(i, j) \triangleq \\
\quad \text{LET } msg \triangleq [destination \mapsto i,\ content \mapsto [type \mapsto \text{"Lookup"},\ node \mapsto j]] \\
\quad \text{IN } \ \wedge\ Status[i] \neq \text{"Dead"} \wedge i \neq j \\
\qquad\quad \wedge\ MessagePool' = MessagePool \cup \{msg\} \\
\qquad\quad \wedge\ \text{UNCHANGED } \langle Status,\ RoutingTables,\ LeafSets,\ Probing, \\
\qquad\qquad\qquad\qquad Failed,\ Leases,\ Grants,\ ToJoin\rangle
\end{array}
$$

Definition 18 (Action Route Lookup)**.**

$$
\begin{array}{l}
RouteLookup(i, j) \triangleq \\
\quad Status[i] \neq \text{"Dead"} \wedge \exists\, m \in MessagePool : \\
\qquad \wedge\ m.content.type = \text{"Lookup"} \wedge m.destination = i \\
\qquad \wedge\ m.content.node = j \wedge \neg Covers(LeafSets[i], j) \\
\qquad \wedge\ \text{LET } nh \triangleq NextHop(i, j) \\
\qquad\qquad\quad msg1 \triangleq [destination \mapsto nh,\ content \mapsto m.content] \\
\qquad\qquad\quad msg2 \triangleq [destination \mapsto i, \\
\qquad\qquad\qquad\qquad content \mapsto [type \mapsto \text{"NoLegalRoute"},\ key \mapsto j]] \\
\qquad\quad \text{IN } \ \wedge\ MessagePool' = (MessagePool \setminus \{m\}) \cup \\
\qquad\qquad\qquad\qquad\qquad \text{IF } nh \neq i \text{ THEN } \{msg1\} \text{ ELSE } \{msg2\} \\
\qquad\qquad\quad \wedge\ \text{UNCHANGED } \langle Status,\ RoutingTables,\ LeafSets,\ Probing, \\
\qquad\qquad\qquad\qquad\qquad Failed,\ Leases,\ Grants,\ ToJoin\rangle
\end{array}
$$

Definition 19 (Action Deliver Lookup)**.**

$$
\begin{array}{l}
DeliverLookup(i, j) \triangleq \\
\quad \wedge\ Status[i] = \text{"Ready"} \wedge Covers(LeafSets[i], j) \\
\quad \wedge\ \exists\, m \in MessagePool : \\
\qquad \wedge\ m.content.type = \text{"Lookup"} \wedge m.destination = i \\
\qquad \wedge\ m.content.node = j \wedge MessagePool' = (MessagePool \setminus \{m\}) \\
\quad \wedge\ \text{UNCHANGED } \langle Status,\ LeafSets,\ RoutingTables,\ Probing,\ Leases, \\
\qquad\qquad\qquad\qquad Grants,\ Failed,\ ToJoin\rangle
\end{array}
$$

Join Handling

The join handling phase of the protocol is modeled in four actions. The first action is *Join*, where a new node gets assigned the unused identifier j and wishes to join the network. The node changes its status to Waiting, and issues a join request message to some Ready node i. As can be seen in the definition, $Join(j, i)$ is an action executed by node j, but it refers to the local variable $Status[i]$ of a different node i. This is the only place in the specification where this is allowed; it is assumed that j has some means of obtaining the address of some Ready node on the network, for example through the designated server that is also responsible for assigning j its ID.

A node i that receives a join request from a node j that it does not cover has to forward this request to the next node k along the way. This is modeled by the action *RouteJoinRequest*.

If a Ready node i receives a join request message from a node j that it covers, it can only handle this join request if it is not handling any other join requests at the moment, i.e. if its to-join field is clear. In this case, i handles the join request by setting its to-join field to j, adding j to its leaf set, and responding to j with a join reply message. Node i attaches its leaf set to the join reply message. This is modeled by the action *ReceiveJoinRequest*.

Finally, in action *ReceiveJoinReply*, a Waiting node i receives a reply to its join request. Node i adds all content of the attached leaf set to its own leaf set. Note that, in case of an overflow, only the L closest nodes to i from the left and right sides will remain in i's new leaf set. Node i now enters the probing phase by sending probe messages to those nodes that have been successfully added to its leaf set, i.e. those nodes that are sufficiently close to it.

The TLA$^+$ expression $ProbeSet(i, ls, a, b)$ is simply a set of probe messages from i to each node in a set of nodes b. Attached to each probe message is the leaf set ls and a set of nodes a suspected by i to be failed nodes.

Definition 20 (Action Join)**.**

$$
\begin{array}{l}
Join(j, i) \triangleq \\
\quad \text{LET } msg \triangleq [destination \mapsto i, \\
\qquad\qquad content \mapsto [type \mapsto \text{"JoinRequest"}, rt \mapsto EmptyRT, \\
\qquad\qquad\qquad node \mapsto j]] \\
\quad \text{IN} \quad \land\ Status[j] = \text{"Dead"} \land Status[i] = \text{"Ready"} \\
\qquad\quad \land\ Status' = [Status \text{ EXCEPT } ![j] = \text{"Waiting"}] \\
\qquad\quad \land\ MessagePool' = MessagePool \cup \{msg\} \\
\qquad\quad \land\ \text{UNCHANGED } \langle RoutingTables, LeafSets, Probing, Failed, Leases, \\
\qquad\qquad\qquad\qquad Grants, ToJoin\rangle
\end{array}
$$

Definition 21 (Action Route Join Request)**.**

$$
\begin{array}{l}
RouteJoinRequest(i, j) \triangleq \\
\quad Status[i] \neq \text{"Dead"} \land \exists\, m \in MessagePool : \\
\qquad \text{LET } nh \triangleq NextHop(i, j) \\
\qquad\qquad msg1 \triangleq [destination \mapsto nh,\ content \mapsto \\
\qquad\qquad\qquad [type \mapsto \text{"JoinRequest"}, \\
\qquad\qquad\qquad\ rt \mapsto AddToRT(RTContent(RoutingTables[i]), \\
\qquad\qquad\qquad\qquad m.content.rt,\ i),\ node \mapsto j]] \\
\qquad\qquad msg2 \triangleq [destination \mapsto i, \\
\qquad\qquad\qquad content \mapsto [type \mapsto \text{"NoLegalRoute"},\ key \mapsto j]] \\
\qquad \text{IN} \quad \land\, m.content.type = \text{"JoinRequest"} \land m.destination = i \\
\qquad\qquad \land\, m.content.node = j \land \neg Covers(LeafSets[i], j) \\
\qquad\qquad \land\, MessagePool' = (MessagePool \setminus \{m\}) \cup \\
\qquad\qquad\quad \text{IF } nh \neq i \text{ THEN } \{msg1\} \text{ ELSE } \{msg2\} \\
\qquad\qquad \land\, \text{UNCHANGED } \langle Status,\ RoutingTables,\ LeafSets,\ Probing, \\
\qquad\qquad\qquad\qquad Failed,\ Leases,\ Grants,\ ToJoin\rangle
\end{array}
$$

Definition 22 (Action Receive Join Request)**.**

$$
\begin{array}{l}
ReceiveJoinRequest(i) \triangleq \\
\quad Status[i] = \text{"Ready"} \land ToJoin[i] = i \land \exists\, m \in MessagePool : \\
\qquad \land\, m.content.type = \text{"JoinRequest"} \land m.destination = i \\
\qquad \land\, Covers(LeafSets[i],\ m.content.node) \\
\qquad \land\, \text{LET } cont \triangleq [type \mapsto \text{"JoinReply"},\ rt \mapsto m.content.rt, \\
\qquad\qquad\qquad ls \mapsto LeafSets[i]] \\
\qquad\qquad msg \triangleq [destination \mapsto m.content.node,\ content \mapsto cont] \\
\qquad\quad \text{IN} \quad \land\, MessagePool' = (MessagePool \setminus \{m\}) \cup \{msg\} \\
\qquad\qquad \land\, ToJoin' = [ToJoin \text{ EXCEPT } ![i] = m.content.node] \\
\qquad\qquad \land\, LeafSets' = [LeafSets \text{ EXCEPT } ![i] = \\
\qquad\qquad\qquad\qquad AddToLS(\{m.content.node\},\ @)] \\
\qquad\qquad \land\, \text{UNCHANGED } \langle Status,\ RoutingTables,\ Probing,\ Failed, \\
\qquad\qquad\qquad\qquad Leases,\ Grants\rangle
\end{array}
$$

Definition 23 (Action Receive Join Reply)**.**

$$
\begin{array}{l}
ReceiveJoinReply(i) \triangleq Status[i] = \text{"Waiting"} \wedge \exists\, m \in MessagePool : \\
\quad \wedge\, m.content.type = \text{"JoinReply"} \wedge m.destination = i \\
\quad \wedge\, \text{LET}\ nrt \triangleq AddToRT(LeafSetContent(m.content.ls) \\
\qquad\qquad\qquad\qquad \cup\, RTContent(m.content.rt),\ RoutingTables[i],\ i) \\
\qquad\quad nls \triangleq AddToLS(LeafSetContent(m.content.ls),\ LeafSets[i]) \\
\qquad\quad prb \triangleq LeafSetContent(nls) \setminus \{i\} \\
\qquad\quad msg \triangleq ProbeSet(i,\ nls,\ \{\},\ prb) \\
\qquad \text{IN} \quad \wedge\, RoutingTables' = [RoutingTables\ \text{EXCEPT}\ ![i] = nrt] \\
\qquad\qquad\quad \wedge\, LeafSets' \quad = [LeafSets\ \text{EXCEPT}\ ![i] = nls] \\
\qquad\qquad\quad \wedge\, Probing' = [Probing\ \text{EXCEPT}\ ![i] = prb] \\
\qquad\qquad\quad \wedge\, MessagePool' = (MessagePool \setminus \{m\}) \cup msg \\
\qquad\qquad\quad \wedge\, \text{UNCHANGED}\ \langle Status,\ Leases,\ Grants,\ ToJoin,\ Failed \rangle
\end{array}
$$

Probing

As mentioned above, a node begins the probing phase as soon as it has received its join reply message. A node always attaches its own leaf set to any probe message that it sends out. Nodes that receive a probe message may also start probing new nodes that they discover in the attached leaf set of the message.

Therefore, two main actions describe the probing phase. In the action *ReceiveProbe*, a node i receives a probe message from a node j. Node i adds j to its leaf set and routing table, and looks in the attached leaf set for new nodes that are potential members of its own leaf set. That is, i looks for nodes that are close enough to it to be in its leaf set, but are not. Node i then sends probe messages to those new nodes.

Action *ReceiveProbeReply* describes what happens when a node i receives a reply to its probe from node j. Similarly to the action *ReceiveProbe*, node i adds node j to its leaf set and routing table, and then looks in the message attachment for new nodes to probe. Additionally, node i also removes j from the set of nodes it is probing. Node i then checks to see if it has finished its probing phase. If node i is a Waiting node and there are no more nodes left for i to probe, node i changes its status to OK and starts a new phase, which is the lease exchange phase. Node i sends lease request messages to its left and right leaf set neighbors.

Definition 24 (Action Receive Probe)**.**

$$
\begin{array}{l}
ReceiveProbe(i) \triangleq \\
\quad \land\ Status[i] = \text{"Ready"} \lor LeafSets[i] \neq EmptyLS(i) \\
\quad \land\ \exists\, m \in MessagePool : \\
\qquad \land\ m.content.type = \text{"Probe"} \land m.destination = i \\
\qquad \land\ \text{LET } j \triangleq m.content.node \\
\qquad\qquad\quad nf \triangleq Failed[i] \setminus \{j\} \\
\qquad\qquad\quad nls1 \triangleq AddToLS(\{j\},\ LeafSets[i]) \\
\qquad\qquad\quad nls2 \triangleq AddToLS((LeafSetContent(m.content.ls) \setminus nf),\ nls1) \\
\qquad\qquad\quad pm \triangleq LeafSetContent(nls2) \\
\qquad\qquad\quad prb1 \triangleq (LeafSetContent(nls1) \cap m.content.failed) \setminus \{i\} \\
\qquad\qquad\quad prb2 \triangleq pm \setminus LeafSetContent(nls1) \\
\qquad\qquad\quad prb3 \triangleq (prb1 \cup prb2) \setminus (Probing[i] \cup nf \cup \{j\}) \\
\qquad\qquad\quad cont \triangleq [type \mapsto \text{"ProbeReply"},\ node \mapsto i, \\
\qquad\qquad\qquad\qquad ls \mapsto LeafSets[i],\ failed \mapsto nf] \\
\qquad\quad \text{IN} \quad \land\ Failed' = [Failed \text{ EXCEPT } ![i] = nf] \\
\qquad\qquad\qquad \land\ RoutingTables' = [RoutingTables \text{ EXCEPT } ![i] = \\
\qquad\qquad\qquad\qquad\qquad\qquad AddToRT(\{j\},\ @,\ i)] \\
\qquad\qquad\qquad \land\ LeafSets' = [LeafSets \text{ EXCEPT } ![i] = nls1] \\
\qquad\qquad\qquad \land\ Probing' = [Probing \text{ EXCEPT } ![i] = @ \cup prb3] \\
\qquad\qquad\qquad \land\ MessagePool' = (MessagePool \\
\qquad\qquad\qquad\quad \cup \{[destination \mapsto j,\ content \mapsto cont]\} \\
\qquad\qquad\qquad\quad \cup ProbeSet(i,\ nls1,\ nf,\ prb3)) \setminus \{m\} \\
\quad \land\ \text{UNCHANGED } \langle Leases,\ Status,\ Grants,\ ToJoin \rangle
\end{array}
$$

Definition 25 (Action Receive Probe Reply)**.**

$$
\begin{array}{l}
ReceiveProbeReply(i) \triangleq \\
\quad Status[i] \neq \text{"Dead"} \wedge \exists\, m \in MessagePool : \\
\qquad \wedge\ m.content.type = \text{"ProbeReply"} \wedge m.destination = i \\
\qquad \wedge\ \text{LET}\ j \triangleq m.content.node \\
\qquad\qquad\quad ls \triangleq m.content.ls \\
\qquad\qquad\quad nf \triangleq Failed[i] \setminus \{j\} \\
\qquad\qquad\quad nls1 \triangleq AddToLS(\{j\},\ LeafSets[i]) \\
\qquad\qquad\quad nls2 \triangleq AddToLS((LeafSetContent(ls) \setminus nf),\ nls1) \\
\qquad\qquad\quad pm \triangleq LeafSetContent(nls2) \\
\qquad\qquad\quad prb1 \triangleq (LeafSetContent(nls1) \cap m.content.failed) \\
\qquad\qquad\qquad\qquad \setminus (Probing[i] \cup nf \cup \{i\}) \\
\qquad\qquad\quad prb2 \triangleq pm \setminus (LeafSetContent(nls1) \\
\qquad\qquad\qquad\qquad \cup\ Probing[i] \cup nf \cup prb1) \\
\qquad\qquad\quad prb3 \triangleq (Probing[i] \cup prb1 \cup prb2) \setminus \{j\} \\
\qquad\qquad\quad fin \triangleq Status[i] = \text{"Waiting"} \wedge prb3 = \{\} \\
\qquad\qquad\quad msg1 \triangleq [destination \mapsto LeftNeighbor(nls1), \\
\qquad\qquad\qquad\qquad content \mapsto [type \mapsto \text{"LeaseRequest"},\ node \mapsto i]] \\
\qquad\qquad\quad msg2 \triangleq [destination \mapsto RightNeighbor(nls2), \\
\qquad\qquad\qquad\qquad content \mapsto [type \mapsto \text{"LeaseRequest"},\ node \mapsto i]] \\
\qquad\ \ \text{IN}\ \ \wedge\ RoutingTables' = \\
\qquad\qquad\qquad [RoutingTables\ \text{EXCEPT}\ ![i] = AddToRT(\{j\},\ @,\ i)] \\
\qquad\qquad\quad \wedge\ LeafSets' = [LeafSets\ \text{EXCEPT}\ ![i] = nls1] \\
\qquad\qquad\quad \wedge\ Failed' = [Failed\ \text{EXCEPT}\ ![i] = \text{IF}\ fin\ \text{THEN}\ \{\}\ \text{ELSE}\ nf] \\
\qquad\qquad\quad \wedge\ Probing' = [Probing\ \text{EXCEPT}\ ![i] = prb3] \\
\qquad\qquad\quad \wedge\ Status' = [Status\ \text{EXCEPT}\ ![i] = \\
\qquad\qquad\qquad\qquad \text{IF}\ fin\ \text{THEN}\ \text{"OK"}\ \text{ELSE}\ @] \\
\qquad\qquad\quad \wedge\ MessagePool' = (MessagePool \\
\qquad\qquad\qquad \cup\ ProbeSet(i,\ nls1,\ nf,\ prb1) \cup ProbeSet(i,\ nls1,\ nf,\ prb2) \\
\qquad\qquad\qquad \cup\ \text{IF}\ fin\ \text{THEN}\ \{msg1,\ msg2\}\ \text{ELSE}\ \{\}) \setminus \{m\} \\
\qquad\qquad\quad \wedge\ \text{UNCHANGED}\ \langle Leases,\ Grants,\ ToJoin\rangle
\end{array}
$$

Lease Exchange

The lease exchange phase begins when a Waiting node i receives the last probe reply message, changes its status to OK, and sends lease request messages to its left and right leaf set neighbors.

A Ready/OK node i that receives a lease request grants the lease only if the request comes from a leaf set neighbor. This is described in the action *ReceiveLeaseRequest*.

There is a possibility that a node i issues a lease request to its leaf set neighbor j, but by the time the request arrives at j, i is no longer j's leaf set neighbor. This can happen if a new node k between i and j on the ring

starts the join process through node j, becoming its new leaf set neighbor. In this case, j does not grant i's lease request, and i must autonomously repeat the request at a later time, when it may have updated its leaf set neighbor. This case is handled by the action *RequestLease*, where a node i autonomously requests a lease from one of its leaf set neighbors.

Finally, a node that receives a lease reply from its neighbor granting it the lease, responds by granting its neighbor a lease in return. If the node has been granted leases from both its neighbors, it switches from OK to Ready. This is described by the action *ReceiveLeaseReply*.

Definition 26 (Action Receive Lease Request)**.**

$$
\begin{array}{l}
\mathit{ReceiveLeaseRequest}(i) \triangleq \\
\quad \mathit{Status}[i] \in \{\text{“OK”}, \text{“Ready”}\} \land \exists\, m \in \mathit{MessagePool} : \\
\quad\quad \textsc{let}\ \mathit{gr} \triangleq m.\mathit{content}.\mathit{node} \in \{\mathit{LeftNeighbor}(\mathit{LeafSets}[i]), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathit{RightNeighbor}(\mathit{LeafSets}[i])\} \\
\quad\quad\quad\quad \mathit{msg} \triangleq [\mathit{destination} \mapsto m.\mathit{content}.\mathit{node}, \\
\quad\quad\quad\quad\quad\quad \mathit{content} \mapsto [\mathit{type} \mapsto \text{“LeaseReply”}, \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{ls} \mapsto \mathit{LeafSets}[i], \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{grant} \mapsto \mathit{gr}]] \\
\quad\quad \textsc{in}\ \ \land\ m.\mathit{content}.\mathit{type} = \text{“LeaseRequest”} \land m.\mathit{destination} = i \\
\quad\quad\quad\quad \land\ \mathit{Grants}' = [\mathit{Grants}\ \textsc{except}\ ![i] = \\
\quad\quad\quad\quad\quad\quad\quad\quad \textsc{if}\ \mathit{gr}\ \textsc{then}\ @ \cup \{m.\mathit{content}.\mathit{node}\}\ \textsc{else}\ @] \\
\quad\quad\quad\quad \land\ \mathit{MessagePool}' = (\mathit{MessagePool} \setminus \{m\}) \cup \{\mathit{msg}\} \\
\quad\quad\quad\quad \land\ \textsc{unchanged}\ \langle \mathit{Status}, \mathit{RoutingTables}, \mathit{LeafSets}, \mathit{Probing}, \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Failed}, \mathit{Leases}, \mathit{ToJoin} \rangle
\end{array}
$$

Definition 27 (Action Request Lease)**.**

$$
\begin{array}{l}
\mathit{RequestLease}(i) \triangleq \\
\quad \textsc{let}\ \mathit{ln} \triangleq \mathit{LeftNeighbor}(\mathit{LeafSets}[i]) \\
\quad\quad\quad \mathit{rn} \triangleq \mathit{RightNeighbor}(\mathit{LeafSets}[i]) \\
\quad\quad\quad \mathit{msg}(d) \triangleq [\mathit{destination} \mapsto d, \\
\quad\quad\quad\quad\quad\quad \mathit{content} \mapsto [\mathit{type} \mapsto \text{“LeaseRequest”}, \mathit{node} \mapsto i]] \\
\quad \textsc{in} \\
\quad \land\ \mathit{Status}[i] = \text{“OK”} \\
\quad \land\ \neg(\mathit{ln} \in \mathit{Leases}[i] \land \mathit{rn} \in \mathit{Leases}[i]) \\
\quad \land\ \mathit{MessagePool}' = \mathit{MessagePool} \\
\quad\quad\quad \cup\ \textsc{if}\ \mathit{ln} \notin \mathit{Leases}[i]\ \textsc{then}\ \{\mathit{msg}(\mathit{ln})\}\ \textsc{else}\ \{\} \\
\quad\quad\quad \cup\ \textsc{if}\ \mathit{rn} \notin \mathit{Leases}[i]\ \textsc{then}\ \{\mathit{msg}(\mathit{rn})\}\ \textsc{else}\ \{\} \\
\quad \land\ \textsc{unchanged}\ \langle \mathit{Status}, \mathit{LeafSets}, \mathit{RoutingTables}, \mathit{Probing}, \mathit{Failed}, \\
\quad\quad\quad\quad\quad \mathit{Leases}, \mathit{Grants}, \mathit{ToJoin} \rangle
\end{array}
$$

Definition 28 (Action Receive Lease Reply)**.**

$$
\begin{array}{l}
ReceiveLeaseReply(i) \triangleq \\
\quad Status[i] \in \{\text{"Ready"}, \text{"OK"}\} \land \exists\, m \in MessagePool : \\
\qquad \land\ m.content.type = \text{"LeaseReply"} \land m.destination = i \\
\qquad \land\ \text{LET}\ ln \triangleq LeftNeighbor(LeafSets[i]) \\
\qquad\qquad\quad rn \triangleq RightNeighbor(LeafSets[i]) \\
\qquad\qquad\quad nl \triangleq \text{IF}\ m.content.grant \\
\qquad\qquad\qquad\qquad \text{THEN}\ Leases[i] \cup \{m.content.ls.node\} \\
\qquad\qquad\qquad\qquad \text{ELSE}\ Leases[i] \\
\qquad\qquad\quad fin \triangleq ln \in nl \land rn \in nl \land Status[i] = \text{"OK"} \\
\qquad\qquad\quad msg(d) \triangleq [destination \mapsto d, \\
\qquad\qquad\qquad\qquad content \mapsto [type \mapsto \text{"LeaseReply"}, \\
\qquad\qquad\qquad\qquad\qquad\qquad ls \mapsto LeafSets[i],\ grant \mapsto \text{TRUE}]] \\
\qquad\quad \text{IN}\quad \land\ m.content.ls.node \in \{ln, rn\} \\
\qquad\qquad\quad \land\ Leases' = [Leases\ \text{EXCEPT}\ ![i] = nl] \\
\qquad\qquad\quad \land\ ToJoin' = [ToJoin\ \text{EXCEPT}\ ![i] = \\
\qquad\qquad\qquad\qquad \text{IF}\ ToJoin[i] = m.content.ls.node\ \text{THEN}\ i \\
\qquad\qquad\qquad\qquad \text{ELSE}\ @] \\
\qquad\qquad\quad \land\ Status' = [Status\ \text{EXCEPT}\ ![i] = \text{IF}\ fin\ \text{THEN}\ \text{"Ready"} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ @] \\
\qquad\qquad\quad \land\ Grants' = [Grants\ \text{EXCEPT}\ ![i] = \text{IF}\ fin\ \text{THEN}\ @ \cup \{ln, rn\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ @] \\
\qquad\qquad\quad \land\ MessagePool' = \text{IF}\ Fin\ \text{THEN}\ (MessagePool \setminus \{m\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cup \{msg(ln), msg(rn)\} \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ (MessagePool \setminus \{m\}) \\
\qquad\qquad\quad \land\ \text{UNCHANGED}\ \langle LeafSets, RoutingTables, Probing, Failed\rangle
\end{array}
$$

As can be seen from the above definitions, a dead node has one capability, which is to turn Waiting and issue a request to join the network. All non-Dead nodes can initiate lookups and route lookup and join request messages. Only Ready nodes handle lookup and join request messages. All non-Dead nodes can exchange probe messages except Waiting nodes that have not received a reply to their join request. Even after finishing the probing phase, a Ready/OK node may still receive probes from other nodes, triggering a new probing phase; these additional probing phases do not affect the Ready/OK status of a node but only the content of its leaf set. Only Ready/OK nodes exchange lease request and reply messages.

This concludes the TLA$^+$ specification of LuPastry$^+$. In the next chapter, I present the key differences between LuPastry$^+$ and the original specification and proof of LuPastry by Lu.

Chapter 5

Improvements in LuPastry$^+$ to LuPastry

LuPastry$^+$ is an improved version of the LuPastry specification in TLA$^+$, accompanied with a complete proof of correct delivery of lookup messages. Section 5.1 starts with an analysis of Lu's partial proof of correct delivery for LuPastry, pointing out several counterexamples to assumptions used in Lu's proof, and highlighting the need for the new, complete proof I present in this thesis. In Section 5.2, I describe the changes I introduce to Lu's TLA$^+$ specification of LuPastry that make both the specification and the final correctness proof more modular, readable and concise. Finally, Section 5.3 gives an overview of the structure of the new correctness proof.

5.1 Lu's Partial Proof of Correct Delivery

Figure 5.1 shows the structure of the original TLA$^+$ specification and partial correctness proof of LuPastry by Lu [27].[1]

1. At the bottom-most layer is Lu's TLA$^+$ specification of LuPastry.

2. The *Arithmetic and Theory* layer consists of a large number of unproven mathematical assumptions, mostly stated as TLA$^+$ lemmas where the proof is omitted.

3. The *Leaf Set Properties* layer consists of a large number of unproven assumptions on the leaf set data structure, which is the main relevant data structure for proving correct delivery.

4. The *Reduction to Invariants* layer contains the main theorems that reduce the safety property of correct delivery to a set of 50 claimed invariants

[1] Visualization and names assigned to proof layers are my own. Line counts represent the total line counts of the individual TLA$^+$ files, which may include whitepace and comments. However, I exclude all trailing comments in a file.

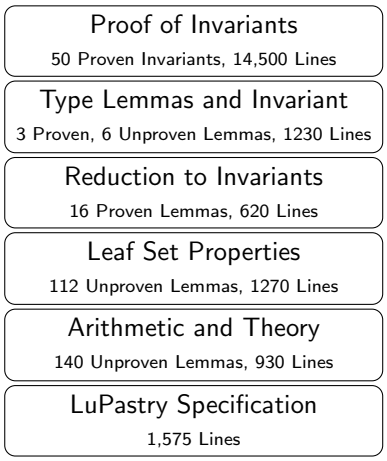


Figure 5.1 – Structure of the original LuPastry specification and proof.

formulated by Lu. The lemmas and theorems in this layer are proven in TLA$^+$, but using the underlying unproven mathematical assumptions in the Arithmetic and Theory layer, as well as unproven assumptions about the leaf set data structure in the Leaf Set Properties layer.

5. The *Type Lemmas and Invariant* layer is concerned with the types of certain TLA$^+$ operators, as well as the types of all specification variables in *vars* which are given by the *type invariant*. This layer contains both unproven assumptions, and some lemmas proven relying on other unproven assumptions.

6. Finally, the *Proof of Invariants* layer contains a complete TLA$^+$ proof of the 50 claimed invariants of the protocol. Naturally, this layer of the proof relies heavily on the unproven assumptions about the leaf set data structure and ring arithmetic.

The fact that Lu's proof relied on a large number of unproven assumptions is due to the sheer size of the proof, and the lack of maturity of the TLA$^+$ proof assistant at that time. For example, a large part of Lu's proof was written before the SMT back-end of TLA$^+$ was developed, and therefore it was a sensible decision at the time to make a separation between lemmas about data structures, which often required theory reasoning and remained unproven, and proven invariants relating to the behavior of the system.

I have attempted to prove Lu's unproven assumptions, and found counterexamples to many of them, such as arithmetic assumptions ignoring border cases. Moreover, several assumptions were stated but not actually used later in the proof. My analysis of Lu's proof can be summarized in the following points.

1. Many assumptions in the Arithmetic and Theory layer were not used later in the proof. Also, many assumptions were stated in such a way that made them difficult to prove in TLA$^+$ using the available encodings from TLA$^+$ to the back-end provers (for example, by relying heavily on the division operator). Some assumptions ignored border cases, which means they had to be reformulated (i.e. weakened). I did not attempt to reformulate and prove all the lemmas in this layer, since it seemed more efficient to start from the top-level proof and go back to prove only as many arithmetic facts as needed. Moreover, as will be discussed in Section 5.2, I have changed some definitions in the LuPastry specification in order to make arithmetic reasoning much simpler, for example, by replacing division by multiplication whenever possible.

2. The Leaf Set Properties layer contains 112 unproven assumptions about the leaf set data structure. Upon examining these assumptions, I could prove only 21 directly. I discovered that 34 assumptions were unneeded later in the proof. The rest of the assumptions were incorrect. I attempted to reformulate and prove all incorrect assumptions in this layer. For six of the assumptions, it was not possible to find a correct formulation; i.e. the assumption could not be weakened in such a way that would still make it useful for the proof. See Figure 5.2. As will be discussed later, the new, complete proof required a new layer of leaf set lemmas in which I include lemmas from Lu that could be reformulated and proven, as well as many new lemmas that were required in the top-level proof.

3. The theorems in the Reduction to Invariants layer rely heavily on the unproven arithmetic and leaf set assumptions, and reduce correct delivery to a set of 50 claimed invariants by Lu. As I will point out in what follows, I have discovered a counterexample to one of these claimed invariants. Therefore, my conclusion was that a new reduction proof was needed, reducing correct delivery to a set of proven correctness invariants, where both the reduction proof and the proof of the correctness invariants would rely only on proven facts about arithmetic and the leaf set data structure.

4. The Type Lemmas and Invariant layer also contains some unproven assumptions. More importantly, it contains only a few lemmas and is missing lemmas about the types of most operators and functions of the specification. Because TLA$^+$ is an untyped language, a significant part of the proof of any TLA$^+$ proof obligation is proving types for the TLA$^+$ expressions appearing in that obligation. Therefore, a rigorous, complete proof of correctness would require a much more comprehensive layer of type lemmas. Also, this layer would need to appear much earlier in the proof hierarchy, since it is also useful for proving leaf set properties and for the reduction proof.

5. The Proof of Invariants layer is the largest part of Lu's proof, and contains a complete TLA$^+$ proof of 50 claimed invariants, based largely on the

Assumptions	Count
Proven	21
Unneeded	34
False, Weakened	51
False	6
Total	112

Figure 5.2 – Analysis of Lu's unproven assumptions about the leaf set data structure.

unproven and partly wrong assumptions in the previous layers. Using the TLC model checker for TLA$^+$, I discovered a counterexample to one of these claimed invariants. Because of this discovery, and the large size of this layer, I decided against an attempt to re-prove Lu's invariants, opting instead for writing a new correctness proof.

Counterexamples to Lu's Assumptions

In what follows, I give some examples of incorrect assumptions in Lu's proof.[2]

Counterexamples to Leaf Set Assumptions

Consider the following assumption[3], which states that after adding some set of nodes a to a leaf set $ls1$, the right neighbor of the resulting leaf set $ls2$ can only be closer to the leaf set owner i than the original right neighbor of $ls1$.

$$
\begin{aligned}
&\forall\, ls1,\, ls2 \in LeafSet,\, i \in I,\, a \in \textsc{subset}\ I : \\
&\quad i = ls1.node \land ls2 = AddToLS(a,\, ls1) \Rightarrow \\
&\quad\quad ClockwiseDistance(i,\, RightNeighbor(ls2)) \leq \\
&\quad\quad\quad ClockwiseDistance(i,\, RightNeighbor(ls1))
\end{aligned}
$$

This assumption does not hold if the right-hand part of the leaf set is empty. If $ls1.right = \{\}$, then $RightNeighbor(ls1) = i$, and i is closer to itself than to any other node; $ClockwiseDistance(i, i) = 0$. Instead, the assumption needs to be weakened as follows.

$$
\begin{aligned}
&\forall\, ls1,\, ls2 \in LeafSet,\, i \in I,\, a \in \textsc{subset}\ I : \\
&\quad i = ls1.node \land ls1.right \neq \{\} \land ls2 = AddToLS(a,\, ls1) \Rightarrow \\
&\quad\quad ClockwiseDistance(i,\, RightNeighbor(ls2)) \leq \\
&\quad\quad\quad ClockwiseDistance(i,\, RightNeighbor(ls1))
\end{aligned}
$$

[2] Adapting notation for better readability.
[3] See *AddToLSetInvCo* in module *ProofLSetProp* of Lu's LuPastry files, [2].

Similarly, the following assumption states that the leaf set obtained by adding a node k to some leaf set ls, contains the same nodes in ls, and possibly also k. [4]

$$\text{LEMMA } \forall\, ls \in LeafSet,\ k \in I :$$
$$LeafSetContent(AddToLS(\{k\},\ ls)) \setminus \{k\} = LeafSetContent(ls)$$

This is not true: if ls is full (complete), then adding a new node k to it will generally result in some other node being removed from the leaf set, invalidating the claimed equality. There is no obvious weaker alternative of this assumption that is useful to the proof.

Counterexample to Claimed Invariant

Using the TLC model checker, I discovered a counterexample to one of the 50 claimed correctness invariants in Lu's proof.[5]

$$\begin{aligned}
&\forall\, m \in MessagePool : m.content.type = \text{“JoinReply”} \Rightarrow \\
&\quad \text{LET } n \triangleq m.content.node \\
&\quad \text{IN} \quad \land\ ClockwiseDistance(LeftNeighbor(LeafSets[n]),\ n) \\
&\qquad\qquad \leq ClockwiseDistance(LeftNeighbor(m.content.ls),\ n) \\
&\qquad\quad \land\ ClockwiseDistance(n,\ RightNeighbor(LeafSets[n])) \\
&\qquad\qquad \leq ClockwiseDistance(n,\ RightNeighbor(m.content.ls))
\end{aligned}$$

The formula asserts that if some node n sends a *JoinReply* message, then n's current neighbors are closer to it than its neighbors were at the time when the message was sent. This is not true, however, if n's leaf set was empty at the time the message was sent. In case of an empty leaf set, the left and right neighbors of node n are n itself. Any new neighbors of n will be farther away from n than n itself.

5.2 Improvements to the LuPastry Specification

In analyzing Lu's proof, I was able to gain a better insight into the main complexity bottlenecks.

1. Many proof obligations in Lu's top-level proof are repetitive because they require the same reasoning about integer arithmetic, including division, exponentiation and the modulus operator. Many of these obligations fail due to the inability of the SMT back-end to deal with these obligations.

2. Most of Lu's assumptions about the leaf set data structure could only be proven by strengthening the assumptions to include more information

[4] See *AddAndDelete* in module *ProofLSetProp* of Lu's LuPastry files, [2].
[5] See *SemJoinLeafSet* in module *MSPastry* of Lu's LuPastry files, [2].

about how leaf set members are distributed within the leaf set (left or right parts).

3. All proof obligations in Lu's proof that contain CHOOSE fail due to the lack of necessary lemmas, as I will show in what follows.

Exchanging Division for Multiplication

A small change that greatly simplified the task of proving necessary arithmetic assumptions was changing the definition of *RingSize*. In the original specification of LuPastry, the size of the Pastry ring was defined as, $RingSize \triangleq 2^M$. Reasoning about shortest (absolute) distance on the ring requires frequent mention to the value $RingSize \div 2$.

However, the division operator is a rather difficult operator to reason about, especially given the current TLA$^+$ to SMT encoding. In LuPastry$^+$, I define the operator *HalfRingSize* to denote half the size of the ring. *RingSize* is then defined as twice *HalfRingSize*.

$$HalfRingSize \triangleq 2^{(M-1)}$$
$$RingSize \triangleq 2 * HalfRingSize$$

This change eliminates the division operator from many arithmetic assumptions, making them easier to prove automatically using the available SMT back-end.

Isolating Theory Reasoning Using New Operators

I rewrite some TLA$^+$ definitions in LuPastry using new operators that abstract away from arithmetic calculations and reduce the use of TLA$^+$'s CHOOSE operator, which is difficult for back-end provers to reason about and hence restrains automation.

Arithmetic calculations, which mainly involve comparisons between distances between nodes on the ring, appeared so extensively in LuPastry that arithmetic reasoning was frequently needed at the top level of the original proof. For example, one of the most typical subformulas is the comparison,

$$ClockwiseDistance(i, j) \leq ClockwiseDistance(i, k),$$

which appears extensively in LuPastry, as in the definitions of *RightNeighbor*, *AddToLS*, or *Covers* (see Definitions 10, 8 and 12).

In these definitions, the exact distances between nodes are actually irrelevant; the definitions are only concerned with whether some key j lies on the *clockwise path* from key i to key k. However, reasoning about each such comparison still requires that the definition of *ClockwiseDistance* be unfolded.

The standard way of avoiding this problem and simplifying the proof is by renaming the repetitive parts in the definition using new predicates. I

have relied on this idea to make the specification of LuPastry more modular, readable, and concise.

In LuPastry$^+$, I introduce a new predicate called *ClockwiseArc*, where $ClockwiseArc(i, j, k)$ holds if j lies on the clockwise path from i to k. I replace all expressions of the form $ClockwiseDistance(i, j) \leq ClockwiseDistance(i, k)$ in definitions by $ClockwiseArc(i, j, k)$.

Definition 29 (Clockwise Arc)**.**

$$ClockwiseArc(i,\ j,\ k) \triangleq$$
$$ClockwiseDistance(i,\ j) \leq ClockwiseDistance(i,\ k)$$

In the bottom-most Arithmetic and Theory layer of the proof, I then prove once and for all the necessary properties of this relation in TLAPS, using the SMT back-end for arithmetical reasoning. The following are some examples of proven arc properties.

Definition 30 (Examples of Arc Properties)**.**

$$\text{THEOREM } ArcReflexivity \triangleq$$
$$\forall\, x,\ y \in I : ClockwiseArc(x,\ y,\ y) \land ClockwiseArc(x,\ x,\ y)$$

$$\text{THEOREM } ArcAntiSymmetry \triangleq \forall\, x,\ y,\ z \in I :$$
$$\land\ ClockwiseArc(x,\ y,\ z) \land ClockwiseArc(x,\ z,\ y) \Rightarrow y = z$$
$$\land\ ClockwiseArc(x,\ y,\ z) \land ClockwiseArc(y,\ x,\ z) \Rightarrow x = y$$

$$\text{THEOREM } ArcRotation \triangleq \forall\, x,\ y,\ z \in I :$$
$$\land\ x \neq y \land ClockwiseArc(x,\ y,\ z) \Rightarrow ClockwiseArc(y,\ z,\ x)$$
$$\land\ y \neq z \land ClockwiseArc(x,\ y,\ z) \Rightarrow ClockwiseArc(z,\ x,\ y)$$

These theorems are now used heavily in the new proof so that unfolding of the definition of *ClockwiseDistance* (or *ClockwiseArc*) is no longer needed. It is clear that this abstraction helps automate the proof process, since now automatic back-ends like Zenon or Spass without native support for integer arithmetic are able to solve larger steps.

Similarly to *ClockwiseArc*, I define the following predicates for absolute distance. $AbsoluteCloser(x, y, z)$ holds if y is closer to x than z is, in terms of absolute distance. *StrictlyAbsoluteCloser* defines the same notion but in the strict sense.

Definition 31 ((Strictly) Absolute Closer)**.**

$$\mathit{AbsoluteCloser}(x, y, z) \triangleq \mathit{AbsoluteDistance}(y, x) \leq \mathit{AbsoluteDistance}(z, x)$$

$$\mathit{StrictlyAbsoluteCloser}(x, y, z) \triangleq \mathit{AbsoluteDistance}(y, x) < \mathit{AbsoluteDistance}(z, x)$$

A related issue is the extensive use of the CHOOSE operator of TLA$^+$, which is Hilbert's ε-operator for definite choice. As explained in Chapter 3, the TLA$^+$ expression CHOOSE $x \in S : P(x)$ denotes some fixed but arbitrary element x in set S for which the property P holds, if some such x exists. If P holds for no $x \in S$, as in CHOOSE $x \in \mathit{Nat} : x * 0 = 1$, the result of the CHOOSE expression is not specified.

Many LuPastry operators are defined using the CHOOSE operator, such as the operator *RightNeighbor* (see Definition 10). It is unwieldy to reason about operator *RightNeighbor* by unfolding its definition because we would invariably have to show the existence of a node contained in $ls.\mathit{right}$, if $ls.\mathit{right}$ is non-empty, whose distance to $ls.\mathit{node}$ is minimal among all these nodes. Formally, we need the following lemma.

$$\text{LEMMA } \mathit{ClosestExists} \triangleq \forall x \in I, S \subseteq I : S \neq \{\} \Rightarrow \exists y \in S : \forall z \in S : \mathit{ClockwiseDistance}(x, y) \leq \mathit{ClockwiseDistance}(x, z)$$

The proof of this lemma requires induction, and set theory and arithmetic reasoning. Such a lemma does not appear at all in Lu's proof, which is why all proof obligations in Lu's proof that contain CHOOSE fail to be proven using any of the available back-end provers.

Therefore, similar to the operator *ClockwiseArc*, I isolate occurrences of the CHOOSE operator by introducing a new set of *closeness operators*.

Definition 32 (LuPastry$^+$ Closeness Operators)**.**

$$
\begin{aligned}
&\mathit{ClosestFromTheLeft}(x,\ a) \triangleq \\
&\quad \text{IF } a = \{\} \text{ THEN } x \\
&\quad\ \text{ELSE CHOOSE } y \in a : \forall\, z \in a : \mathit{ClockwiseArc}(z,\ y,\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ClosestFromTheRight}(x,\ a) \triangleq \\
&\quad \text{IF } a = \{\} \text{ THEN } x \\
&\quad\ \text{ELSE CHOOSE } y \in a : \forall\, z \in a : \mathit{ClockwiseArc}(x,\ y,\ z)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ClosestNodesFromTheLeft}(x,\ a,\ n) \triangleq \\
&\quad \text{IF } \mathit{Cardinality}(a) \leq n \text{ THEN } a \\
&\quad\ \text{ELSE CHOOSE } \mathit{sub} \in \text{SUBSET } a : \\
&\qquad\qquad \wedge\ \mathit{Cardinality}(\mathit{sub}) = n \\
&\qquad\qquad \wedge\ \forall\, \mathit{in} \in \mathit{sub},\ \mathit{out} \in (a \setminus \mathit{sub}) : \mathit{ClockwiseArc}(\mathit{out},\ \mathit{in},\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{ClosestNodesFromTheRight}(x,\ a,\ n) \triangleq \\
&\quad \text{IF } \mathit{Cardinality}(a) \leq n \text{ THEN } a \\
&\quad\ \text{ELSE CHOOSE } \mathit{sub} \in \text{SUBSET } a : \\
&\qquad\qquad \wedge\ \mathit{Cardinality}(\mathit{sub}) = n \\
&\qquad\qquad \wedge\ \forall\, \mathit{in} \in \mathit{sub},\ \mathit{out} \in (a \setminus \mathit{sub}) : \mathit{ClockwiseArc}(x,\ \mathit{in},\ \mathit{out})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{AbsoluteClosest}(x,\ a) \triangleq \\
&\quad \text{IF } a = \{\} \text{ THEN } x \\
&\quad\ \text{ELSE CHOOSE } y \in a : \forall\, z \in a : \mathit{AbsoluteCloser}(x,\ y,\ z)
\end{aligned}
$$

The expression $\mathit{ClosestFromTheLeft}(x, a)$ evaluates to the closest node y in the set of nodes a to the node x from the left side. If a is empty, then $y = x$. *ClosestFromTheRight* is defined similarly for the right side. The expression $\mathit{ClosestNodesFromTheLeft}(x, a, n)$ evaluates to the n closest nodes in a to x from the left side. If a contains no more than n elements, then the expression evaluates to a. *ClosestNodesFromTheRight* is defined similarly for the right side. Finally, $\mathit{AbsoluteClosest}(x, a)$ returns the closest node y to x in a, in terms of absolute distance.

In LuPastry$^+$, all LuPastry definitions that use CHOOSE are rewritten in terms of closeness operators instead, to abstract away from the CHOOSE expression. For example, the definition of *RightNeighbor* in LuPastry$^+$ is as follows.

$$\mathit{RightNeighbor}(\mathit{ls}) \triangleq \mathit{ClosestFromTheRight}(\mathit{ls.node},\ \mathit{ls.right})$$

For each closeness operator (all of which are defined using CHOOSE, I prove three lemmas.

1. A *choice lemma*, which proves the existence of a value satisfying the characteristic predicate of CHOOSE.

2. A *type lemma*, which proves the type of the operator.
3. An *expansion lemma*, which proves any other relevant properties of the operator, so that they can be used later in the proof without expanding the definition of the operator.

The following would be the choice, type and expansion lemmas for the operator *ClosestFromTheRight*.

```
LEMMA choose_ClosestFromTheRight ≜
  ∀ x ∈ I, a ∈ SUBSET I :
    a ≠ {} ⇒ ∃ y ∈ a : ∀ z ∈ a : ClockwiseArc(x, y, z)

LEMMA type_ClosestFromTheRight ≜
  ∀ x ∈ I, a ∈ SUBSET I : ClosestFromTheRight(x, a) ∈ I

LEMMA def_ClosestFromTheRight ≜
  ∀ x ∈ I, a ∈ SUBSET I :
    ∧ a = {} ⇒ ClosestFromTheRight(x, a) = x
    ∧ a ≠ {} ⇒ ClosestFromTheRight(x, a) ∈ a
    ∧ ∀ y ∈ a : ClockwiseArc(x, ClosestFromTheRight(x, a), y)
```

Of course, type and expansion lemmas are also a useful thing to have for other TLA$^+$ expressions. Type lemmas are crucial for all TLA$^+$ constants, variables, functions and operators. It is useful to have expansion lemmas for all TLA$^+$ functions and operators that occur heavily in the proof, to avoid constantly having to expand their definitions. The following are the type and expansion lemmas for operator *RightNeighbor*.

```
LEMMA type_RightNeighbor ≜
  ∀ ls ∈ LeafSet : RightNeighbor(ls) ∈ I
PROOF BY type_ClosestFromTheRight DEF RightNeighbor, LeafSet

LEMMA def_RightNeighbor ≜
  ∀ ls ∈ LeafSet :
    ∧ ls.right = {} ⇒ RightNeighbor(ls) = ls.node
    ∧ ls.right ≠ {} ⇒ ∧ RightNeighbor(ls) ∈ ls.right
                      ∧ ∀ p ∈ ls.right :
                          ClockwiseArc(ls.node, RightNeighbor(ls), p)
PROOF BY def_ClosestFromTheRight DEF RightNeighbor, LeafSet
```

Defining Additional Leaf Set Properties

By observing Lu's assumptions on the leaf set data structure, I noticed that most of them require some additional properties that were not defined in LuPastry.

Definition 33 (Balanced Leaf Set)**.** *A leaf set is balanced if its left and right parts contain the same number of nodes.*

$IsBalanced(ls) \triangleq$
$\quad Cardinality(ls.left) = Cardinality(ls.right)$

Definition 34 (Proper Leaf Set)**.** *A leaf set is proper if members that are exclusively in the left (right) part are left- (right-)closer to the leaf set owner than other members.*

$IsProper(ls) \triangleq$
$\quad \land \forall x \in ls.left \setminus ls.right,\ y \in ls.right : ClockwiseArc(y,\ x,\ ls.node)$
$\quad \land \forall x \in ls.right \setminus ls.left,\ y \in ls.left : ClockwiseArc(ls.node,\ x,\ y)$

Definition 35 (Organized Leaf Set)**.** *A leaf set is organized if the presence of exclusively left or right members indicates that the leaf set is full.*

$IsOrganized(ls) \triangleq$
$\quad (ls.left \setminus ls.right \neq \{\} \lor ls.right \setminus ls.left \neq \{\}) \Rightarrow IsComplete(ls)$

Effect of New Abstractions on Proof Automation

While the new operators make the TLA$^+$ specification more modular, concise and readable, the most significant gain lies in improving proof automation.

Because all arithmetic comparisons and occurrences of the CHOOSE operator are grouped into only a few definitions, and with the help of a few lemmas on arc properties as well as choice, type and expansion lemmas, I claim that the final correctness proof is much more concise, and therefore more automated, than it would be without these operators. I illustrate this using a simple lemma about adding new nodes to the leaf set data structure, that I prove once with and once without the use of the new operators.

$\text{LEMMA } \forall ls \in LeafSet,\ a \in \text{SUBSET } I : IsProper(AddToLS(a,\ ls))$

Basically, the lemma says that the leaf set obtained by adding some new nodes to a leaf set is proper.

The proof of this lemma using the original definitions of LuPastry consists of 23 interactive proof steps that generate 64 proof obligations. With the new

definitions of LuPastry$^+$, the same proof consists of only 12 interactive proof steps (40 proof obligations). This significant difference comes from the fact that the new operators allow back-end provers to succeed directly on some steps, which would otherwise have to be broken down into further substeps using the original definitions. Already for this simple example there is a 50% reduction in the number of steps, i.e. user interactions.

Other Changes

I introduce some additional changes to the specification that can be summarized as follows.

1. I simplify the definitions of clockwise and absolute distances, which, in turn, simplifies the Arithmetic and Theory layer of the proof. I also eliminate some LuPastry definitions of distance that are unnecessary for the proof. In particular, Lu defined a notion of vector distance or "displacement" from node x to node y; a value that is positive or negative depending on the direction (clockwise or counter-clockwise). Both clockwise and absolute distances were defined in terms of displacement. Instead, I remove the definition of displacement and flatten the definitions of clockwise and absolute distances, as seen in Definitions 3 and 4.

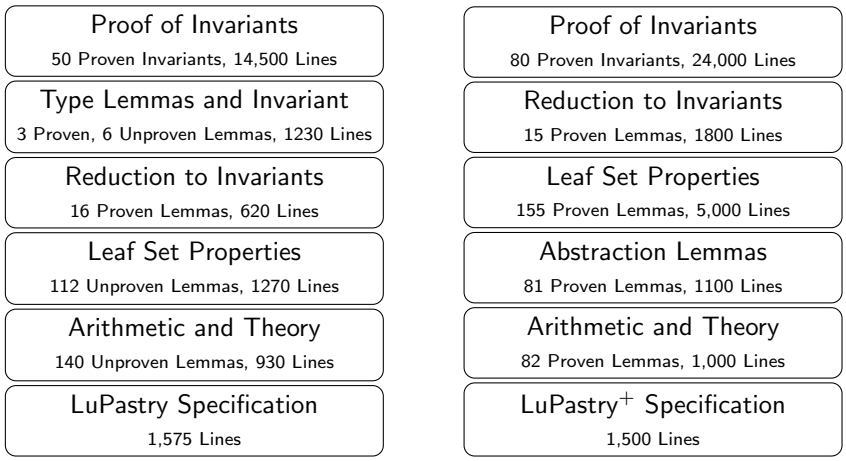
2. I fix some bugs in some of the definitions of the original specification. In particular, definitions relating to message routing and the routing table data structure were not well-defined; they ignored some border cases. This was not discovered by Lu during the proof process, since the proof is partial and is more dependent on the leaf set data structure than the routing process.

3. I modify the probing process so that the node does not attempt to probe itself, which is clearly unnecessary. This simplifies some parts of the proof.

4. Perhaps contrary to convention, I find it more natural to visualize node IDs as increasing in the *clockwise* direction on the ring, and not vice versa. Therefore, in the new specification, and in the explanation of LuPastry in Chapter 4, the definitions are changed from those of Lu so that the notions of "right" and "left" are reversed.

5.3 LuPastry$^+$ Proof Structure

I have written a new, complete proof of correct delivery for LuPastry. The new proof is based on the same assumptions as in Lu's original partial proof. Additionally, I impose the assumption that the node leaf set size L is at least three, $L \geq 3$. The motivation behind this additional assumption is explained in Chapter 6. The structure of the new LuPastry$^+$ proof can be seen on the right side of Figure 5.3, in comparison to Lu's partial proof on the left side.



(a) LuPastry (b) LuPastry$^+$

Figure 5.3 – Structure of the original LuPastry proof versus the new, complete proof.

Refined LuPastry Specification At the bottom layer is the new TLA$^+$ specification of LuPastry, including the improvements discussed in Section 5.2.

Arithmetic and Theory This is the bottom-most layer of the correctness proof, containing all lemmas on ring arithmetic and set theory. All the necessary properties of the distance and closeness operators are proven in this layer.

Abstraction Lemmas This layer of the proof contains all the necessary type lemmas for all TLA$^+$ variables, functions and operators, as well as choice and expansion lemmas, as discussed in Section 5.2. This layer also contains the proof of the overall type invariant of the LuPastry$^+$ variables.

Leaf Set Properties This layer contains a large number of proven lemmas about the leaf set data structure. The lemmas in this layer contain the lemmas reformulated from Lu's proof - as discussed in Section 5.1 - as well as many more I have added as necessary to a rigorous top-level proof.

Reduction to Invariants This layer contains the main theorems reducing the property of correct delivery to 80 correctness invariants I have formulated for LuPastry$^+$.

Proof of Invariants This layer contains proofs for the 80 correctness invariants of LuPastry$^+$.

Because the LuPastry$^+$ proof is rigorous, there was a need for a larger number of invariants than in Lu's proof. Naturally, there is an overlap between the invariants of LuPastry and those of LuPastry$^+$. However, there is no one-to-one correspondence between the two sets of invariants. In particular, while Lu's original proof of correct delivery relies more on the *lease exchange* phase of the protocol, my own proof relies completely on the *probing* process. In fact, in Chapter 7, I adapt the correctness proof of LuPastry$^+$ to a simplified specification of LuPastry$^+$, where the lease exchange phase is entirely omitted from the join process.

Chapter 6

The LuPastry$^+$ Correctness Proof

This chapter presents the TLA$^+$ proof of correct delivery of lookup messages in LuPastry$^+$. I prove that the safety property *correct delivery* is an invariant of the TLA$^+$ specification of LuPastry$^+$. Correct delivery was defined by Lu in [28] as follows.

Invariant 1 (Correct Delivery)**.** *Let i and k be two keys on the ring. If $DeliverLookup(i, k)$ is enabled, then, (1) i is closer to k than any other Ready node $j \neq i$ is, in terms of absolute distance, and (2) the action $DeliverLookup(j, k)$ is not enabled for any key $j \neq i$.*

$$
\begin{aligned}
&CorrectDelivery \triangleq \\
&\quad \forall\, i, j, k \in I : j \neq i \land \text{ENABLED } DeliverLookup(i, k) \Rightarrow \\
&\qquad \land\, Status[j] = \text{"Ready"} \\
&\qquad\qquad \Rightarrow AbsoluteDistance(i, k) \leq AbsoluteDistance(j, k) \\
&\qquad \land\, \neg\text{ENABLED } DeliverLookup(j, k)
\end{aligned}
$$

As per Definition 19 of the action *DeliverLookup*, $DeliverLookup(i, k)$ is *enabled* if and only if i is Ready, its leaf set covers k, and there is a pending lookup message with destination i and key k.

$$
\begin{aligned}
&DeliverLookup(i, k) \triangleq \\
&\quad \land\, Status[i] = \text{"Ready"} \land Covers(LeafSets[i], k) \\
&\quad \land\, \exists\, m \in MessagePool : \\
&\qquad m.content.type = \text{"Lookup"} \land m.content.node = k \\
&\qquad\quad \land\, m.destination = i \land MessagePool' = MessagePool \setminus \{m\} \\
&\quad \land\, \text{UNCHANGED } \langle Status, LeafSets, RoutingTables, Probing, Leases, \\
&\qquad\qquad\qquad Grants, ToJoin, Failed \rangle
\end{aligned}
$$

Currently, TLAPS requires that the ENABLED predicate be unfolded manually in the machine-checked proof. Therefore, my proof uses this stronger definition of correct delivery. The first occurrence of ENABLED at the left hand side of the implication is replaced by the preconditions of the action formula *DeliverLookup*. The second occurrence at the right hand side of the implication is replaced by only one of the preconditions relating to coverage, making the property *StrongCorrectDelivery* stronger than the original property *CorrectDelivery*.

Invariant 2 (Strong Correct Delivery)**.**

$$
\begin{aligned}
&StrongCorrectDelivery \triangleq \forall\, i, j, k \in I : \\
&\quad \land\, Status[i] = \text{"Ready"} \land Status[j] = \text{"Ready"} \\
&\qquad \land\, j \neq i \land Covers(LeafSets[i], k) \\
&\quad \land\, \exists\, m \in MessagePool : m.content.type = \text{"Lookup"} \\
&\qquad \land\, m.content.node = k \land m.destination = i \\
&\Rightarrow \land\, AbsoluteDistance(i, k) \leq AbsoluteDistance(j, k) \\
&\quad \land\, \neg Covers(LeafSets[j], k)
\end{aligned}
$$

Lu also eliminates the keyword ENABLED, but by replacing both of its occurrences with preconditions of action *DeliverLookup*. His partial proof aims at proving the following property which he calls *CorrectCoverage*.

$$
\begin{aligned}
&CorrectCoverage \triangleq \forall\, i, k \in I : \\
&\quad \land\, Status[i] = \text{"Ready"} \\
&\quad \land\, \exists\, m \in MessagePool : \\
&\qquad \land\, m.content.type = \text{"Lookup"} \land m.destination = i \\
&\qquad \land\, m.content.node = k \land Covers(LeafSets[i], k) \\
&\Rightarrow \land\, \forall\, n \in I : Status[n] = \text{"Ready"} \\
&\qquad\qquad \Rightarrow AbsoluteDistance(i, k) \leq AbsoluteDistance(n, k) \\
&\qquad \land\, \neg\exists\, j \in I \setminus \{i\} : \land\, Status[j] = \text{"Ready"} \\
&\qquad\qquad\qquad\qquad \land\, \exists\, m \in MessagePool : \\
&\qquad\qquad\qquad\qquad\quad \land\, m.content.type = \text{"Lookup"} \\
&\qquad\qquad\qquad\qquad\quad \land\, m.destination = j \\
&\qquad\qquad\qquad\qquad\quad \land\, m.content.node = k \\
&\qquad\qquad\qquad\qquad\quad \land\, Covers(LeafSets[j], k)
\end{aligned}
$$

It is clear from the above definitions that *StrongCorrectDelivery* entails *CorrectDelivery*. I show this in the following theorem.

Theorem. *Correct delivery follows from strong correct delivery.*

$$StrongCorrectDelivery \Rightarrow CorrectDelivery$$

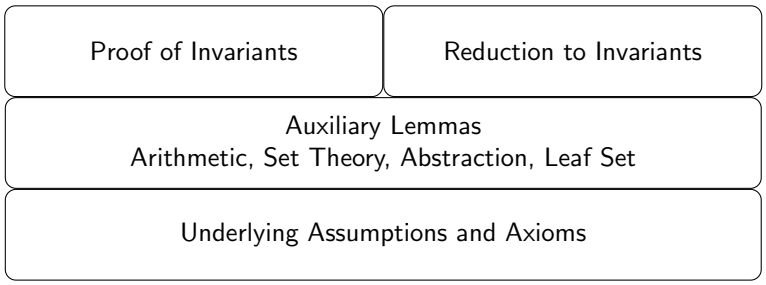
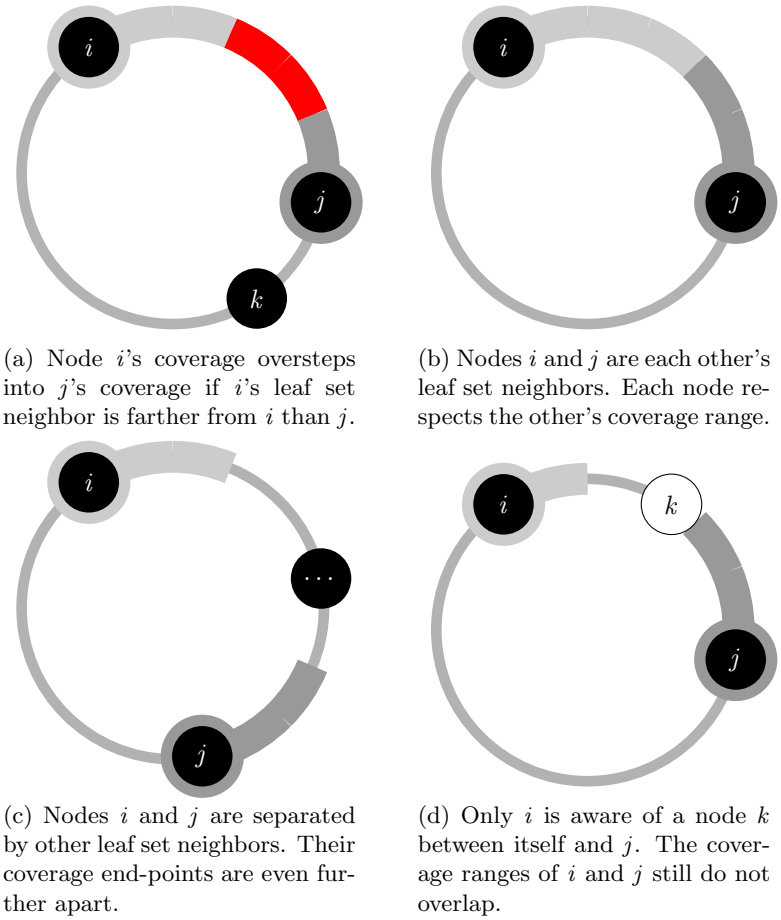


Figure 6.1 – Structure of the LuPastry$^+$ correctness proof.

Proof. Assume that *StrongCorrectDelivery* holds. Consider $i, j, k \in I$, where $j \neq i$ and *DeliverLookup*(i, k) is enabled. Because *DeliverLookup*(i, k) is enabled, i is Ready. It is required to show that (1) if j is Ready, then j is farther from k than i is in terms of absolute distance, and (2) *DeliverLookup*(j, k) is not enabled. The first obligation can be readily discharged by definition of *StrongCorrectDelivery*. It is left to show that *DeliverLookup*(j, k) is not enabled. If j is not Ready, then *DeliverLookup*(j, k) is not enabled by definition of *DeliverLookup*. If j is Ready, then by *StrongCorrectDelivery*, j's leaf set does not cover k. Therefore, *DeliverLookup*(j, k) is not enabled. Therefore, *CorrectDelivery* holds. □

Figure 6.1 shows a more abstract view of the structure of the proof depicted in Figure 5.3b. At the very bottom are a few underlying assumptions and arithmetic axioms, which will be explained in Section 6.2. The middle layer consists of a large number of proven lemmas on arithmetic, set theory, types, the leaf set data structure and coverage among other things, as explained in Chapter 5. The top-level proof reduces (strong) correct delivery to a number of other correctness invariants (right side of the top layer), and proves these invariants with respect to the specification (left side of the top layer). The proof of the invariants alone, without the reduction proof of correct delivery, makes up over 80% of the total proof, and is over 24,000 lines long – it is one and a half times larger than the largest Harry Potter book. Therefore, instead of presenting the proof of the invariants in detail, I will focus on the two most important invariants for the reduction proof: *network stability* and *exclusive coverage*. In Section 6.1, I give the intuition behind these properties and how they are proven to be invariants of LuPastry$^+$. In Section 6.2, I list the assumptions and arithmetic axioms that were necessary for the proof. The proof uses only three unproven arithmetic axioms relating to division, exponentiation and the modulus operator which cannot be proven by the current arithmetic back-ends in TLA$^+$. In Section 6.3, I list all the invariants in the proof.[1] Finally, an outline of the reduction proof, reducing strong correct delivery to the correctness invariants, is given in Section 6.4.

[1]The TLA$^+$ representation of the invariants has been compressed for better readability and compactness.



(a) Node i's coverage oversteps into j's coverage if i's leaf set neighbor is farther from i than j.

(b) Nodes i and j are each other's leaf set neighbors. Each node respects the other's coverage range.

(c) Nodes i and j are separated by other leaf set neighbors. Their coverage end-points are even further apart.

(d) Only i is aware of a node k between itself and j. The coverage ranges of i and j still do not overlap.

Figure 6.2 – Possible arrangements for two Ready nodes i and j and their coverage ranges.

6.1 Intuition

The idea behind the proof is very intuitive. The main task in proving correct delivery is proving that coverages of different Ready nodes are *mutually exclusive*; i.e. that no two Ready nodes i and j cover the same key k.

Proving Exclusive Coverage

Consider two Ready nodes i, j such that $i \neq j$. Both i and j compute their coverage ranges with respect to the content of their leaf set. Figure 6.2 shows the different ways i and j may compute their coverage depending on their leaf sets. It suffices to only consider one side of coverage for each node, e.g. i's right coverage range with j's left coverage range. We can guarantee that the

coverage ranges of i and j do not overlap if i and j "are aware of each other". Consider Figure 6.2a, for example, where node i's right leaf set neighbor is k, but j is right-closer to i than k is. Node i is "unaware" of j, since the closest right leaf set member it knows about is farther than j. Node i computes its coverage to end at the midpoint between itself and k, thereby overstepping its boundary and overlapping with what should be j's coverage range.

Figure 6.2b, on the other hand, shows the situation where i and j are each other's leaf set neighbors. In this case, both i and j compute their coverage ranges with respect to each other, and so the coverages cannot overlap.

Another situation is depicted in Figure 6.2c, where nodes i and j are not each other's leaf set neighbors, but their leaf set neighbors lie between them on the ring. Also in this case the coverages of nodes i and j cannot overlap; the midpoint between i and its right leaf set neighbor is *before* the midpoint between it and j. Similarly, the midpoint between j's left leaf set neighbor and j is *after* the midpoint between i and j. This is another case where i and j are "aware" of each other, even without them being each other's leaf set neighbors.

Finally, it may be that i and j are aware of each other, but there is a node k between i and j that only node i is aware of, as in Figure 6.2d. For example, node k is i's right leaf set neighbor, but j's left leaf set neighbor is i, not k. In this case, j's coverage starts from the midpoint between i and j, but i's coverage extends only up to the midpoint between itself and k. The coverages of i and j do not overlap, but the coverages of j and k do. Because we want to guarantee non-overlapping coverage for all Ready nodes, this situation is benign only for a non-Ready k.

From these examples, it becomes clear that if each Ready node is aware of its right and left Ready neighbors on the ring, we can show non-overlapping coverage. In fact, I observe that it is possible to show non-overlapping coverage for all Ready *and* OK nodes. That is, I prove that in LuPastry$^+$, a Ready/OK node always has its Ready/OK neighbors as members of its leaf set, and is therefore "aware" of them. This observation is the motivation behind the Simplified LuPastry$^+$ protocol I present in Chapter 7, where I show that correct delivery is still guaranteed even if the lease exchange phase of LuPastry$^+$–which helps a node turn from OK to Ready–is eliminated entirely.

More formally, I define *exclusive coverage* for Ready/OK nodes as follows.

$$
\begin{aligned}
&\mathit{ExclusiveCoverage} \triangleq \\
&\quad \forall\, i, j \in \mathit{ReadyOKNodes},\ k \in I : \\
&\qquad i \neq j \wedge i \neq k \wedge j \neq k \\
&\qquad\quad \Rightarrow \neg \mathit{Covers}(\mathit{LeafSets}[i],\ k) \vee \neg \mathit{Covers}(\mathit{LeafSets}[j],\ k)
\end{aligned}
$$

Let a node i be *stable* if its leaf set contains its Ready/OK neighbors on the ring. A LuPastry$^+$ network is a *stable network* if all Ready/OK nodes are stable.

$$
\begin{aligned}
&\mathit{Stable}(i) \triangleq \\
&\quad \land\ \mathit{ClosestFromTheRight}(i,\ \mathit{ReadyOKNodes} \setminus \{i\}) \\
&\qquad\quad \in \mathit{LeafSetContent}(\mathit{LeafSets}[i]) \\
&\quad \land\ \mathit{ClosestFromTheLeft}(i,\ \mathit{ReadyOKNodes} \setminus \{i\}) \\
&\qquad\quad \in \mathit{LeafSetContent}(\mathit{LeafSets}[i])
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{StableNetwork} \triangleq \\
&\quad \forall\, i \in \mathit{ReadyOKNodes} : \mathit{Stable}(i)
\end{aligned}
$$

For example, the ring in Figure 6.3a is stable if nodes $0, 2, 7, 11$ are stable.

It is clear that if we show that *StableNetwork* is an invariant of LuPastry$^+$, we can prove exclusive coverage between all Ready/OK nodes, thereby proving correct delivery.

Proving Network Stability

For a minimum leaf set size $L = 3$, it can be shown that *StableNetwork* is an invariant of LuPastry$^+$ as follows. First, I make the following (proven) remarks about LuPastry$^+$.

1. Because LuPastry$^+$ excludes node failure, all protocol actions that modify a node's leaf set do so through the operation *AddToLS*, as can be seen from Section 4.2. Therefore, nodes are never purposely removed from a leaf set, but a node j may only be evicted from the leaf set of node i through an *AddToLS* operation that results in an overflow; i.e., if the leaf set of i becomes full and j is replaced by another node k that is closer to i than j is.

2. A new node i joins the network through a Ready node r that initially covers it, and so i will remain the closest participating node to r on one side (right or left) until it finishes its join process. Only after i has finished joining and turned Ready can other nodes join the network between r and i. As a result, any Waiting participating node that lies between two Ready/OK neighbors i and j is either the to-join of i or the to-join of j.

3. Let i be a Waiting node, joining through a Ready node r. As soon as r receives i's join request, i is a member of r's leaf set. As soon as i receives r's join reply, r is a member of i's leaf set.

4. Let i and j be two consecutive Ready or OK nodes on the ring (see Figure 6.3b). There can be at most two participating nodes k_1, k_2 between i and j: the to-join nodes of i and j. Any other non-Dead node between i and j must be a Waiting node whose join request has not been picked up by i or j (since they are busy facilitating the joins of k_1 and k_2). Therefore, any Ready/OK node cannot be separated from the Ready/OK node closest to it by more than two nodes.

5. If the leaf set size is three or more, we can ensure that stable i and j remain stable even if new nodes are added to their leaf sets.

From the previous remarks, it is clear that a stable node cannot be destabilized simply by adding new members to its leaf set, since the leaf set is always large enough to accommodate a node's Ready/OK neighbor. Therefore, the main task of proving network stability is to show that as soon as a node turns OK, it is stable, and by turning OK, does not destabilize any previously-stable Ready/OK nodes. This is the most involved and lengthy part of the proof of network stability. A node turns OK as soon as it has finished the probing process and received replies to all its outgoing probes. Let i be a Waiting node in the probing phase joining through some Ready node r, and let j be i's Ready/OK neighbor at the time i turns OK. If we can prove that i has exchanged probes with j before it turned OK, then we can show that at the time when i turns OK, it is stable, and other Ready/OK nodes are aware of it and so remain stable.

There are three different possibilities for the relationship between node j and node i.

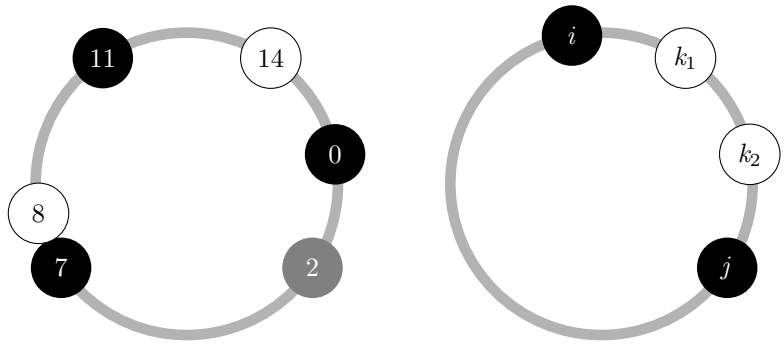
Case 1 Node j is the node responsible for i's joining, $j = r$. In this case, it is guaranteed that j and i are members of each other's leaf sets. Therefore, i is stable when it turns OK, and previously-stable j is still stable when i turns OK.

Case 2 Node j was already a member of r's leaf set when r responded to i's join request. This means that j was in the leaf set which r attached to i in its join reply message. Therefore, i has issued a probe to j and cannot turn OK before it has received a probe reply from j. Therefore, i and j must have exchanged probes before i turned OK.

Case 3 Node j was not a member of r's leaf set when r responded to i's join request. Because r is a stable Ready node, it must be that j was Dead/Waiting. If j were Ready/OK, it would be a Ready/OK neighbor of r and a member of r's leaf set. Therefore, j was a Dead/Waiting node at the time r responded to i's join request, and not a member of r's leaf set. It turned Ready/OK while i was in the probing phase. Let r_2 be the Ready node responsible for j's joining. When j was Dead/Waiting, r and r_2 were each other's Ready/OK neighbors[2], and therefore, r_2 was in the leaf set attached to i's join reply message. Therefore, it must be that i has probed and received a reply from r_2 before turning OK. There are two possibilities.

Case 3a Node i's probe to r_2 reach r_2 before j's join request. In this case, r_2 added i to its leaf set and responded to i's probe, and then

[2]In fact, it is possible that r_2 itself was in the process of joining the network, but in this case the same proof applies recursively on r_2 instead of j. For the sake of this abstract discussion, let us ignore this case and assume r_2 was Ready when j started joining.



(a) This ring is stable if nodes $0, 2, 7, 11$ are stable. Node 0 is stable if its left leaf set contains 11 and its right leaf set contains 2.

(b) For $L \geq 3$, i's leaf set can always accommodate $k_1 = ToJoin[i]$, $k_2 = ToJoin[j]$, and i's Ready/OK neighbor j.

Figure 6.3 – Network stability.

responded to j's join request. This means that i was in the leaf set that r_2 attached to j's join reply. This means that j could not have turned OK before probing and receiving a reply from i.

Case 3b Node j's join request reached r_2 before i's probe did. In this case, j was already in r_2's leaf set, which r_2 attached to the probe reply to i. Therefore, i could not have turned OK before probing and receiving a probe reply from j.

This informal proof of correct delivery is a behavioral-style proof, which focuses mainly on events from a local point of view of a node, and the possible histories of execution leading up to these events. In contrast, the formal TLA$^+$ proof of correct delivery is an assertional-style or invariant-inductive proof, which focuses on global states and uses induction to prove a set of properties to be invariants, i.e. true at every reachable state of the system. Nevertheless, the behavioral argument presented above shows the motivation behind the invariants chosen for the proof and listed in Section 6.3.

6.2 Assumptions and Arithmetic Axioms

The starting point of the proof are assumptions that have to be made about the specification parameters. The proof relies on the following assumptions.

Assumption 1. *NIL is not a valid node.*

$NIL \notin I$

Assumption 2. *A is non-empty set of nodes.*

$A \subseteq I \wedge A \neq \{\}$

Assumption 3. *M and L are positive integers.*

$M \in Nat \setminus \{0\} \wedge L \in Nat \setminus \{0\}$

Assumption 4. *B is a positive integer that is strictly less than M and divides M.*

$B \in Nat \wedge B > 0 \wedge B < M \wedge M\%B = 0$

Assumption 5. *The size of the left (right) leaf set is at least 3.*

$L \geq 3$

In addition, the following are arithmetic facts that cannot be proven by any of the current back-ends in TLA$^+$, and therefore have to be stated as TLA$^+$ axioms without proofs. These are axioms relating to the exponentiation, division and modulus operator respectively.

Axiom 1. *(Exponentiation Properties)*

$$
\begin{aligned}
&\forall\, a,\, b \in Int : \\
&\quad \wedge\, a \neq 0 \Rightarrow a^1 = a \\
&\quad \wedge\, a \neq 0 \Rightarrow \forall\, m,\, n \in Int : a^{(m+n)} = a^m + a^n \\
&\quad \wedge\, b \neq 0 \Rightarrow 0^b = 0 \\
&\quad \wedge\, 0 < a \wedge 0 < b \Rightarrow \forall\, c \in Int : a^{(b*c)} = (a^b)^c \\
&\quad \wedge\, 0 < a \wedge 0 < b \Rightarrow \forall\, c \in Int : b \leq c \Rightarrow a^b \leq a^c
\end{aligned}
$$

Axiom 2. *(Whole Integer Division)*

$\forall\, x,\, y,\, q \in Nat : 0 < y \wedge x = q * y \Rightarrow q = x \div y$

Axiom 3. *(Modulus Properties)*

$\forall\, x,\, y \in Nat : 0 < y \wedge x \geq y \Rightarrow (x\%y) = (x - y)\%y$

6.3 Correctness Invariants

This section contains a list of the proven correctness invariants of LuPastry$^+$. In Section 6.4, it is proved that strong correct delivery follows from these correctness invariants. I denote by *CorrectnessInvariants* the TLA$^+$ conjunction of all the correctness invariants listed here.

The TLA$^+$ proof files reduce correct delivery to 80 correctness invariants. In this presentation, however, many of these invariants have been clustered together or formulated in a more compact way for readability, resulting in fewer formulas in total. In the TLA$^+$ files, it is sometimes much easier to prove a large number of small formulas than a smaller number of larger formulas. This is particularly true for Pastry, because of the symmetry between the "left" and "right" components of the proof. Many invariants consist of a right and left component; for example, the same invariant may need to be proven for both the right and left Ready/OK neighbor of a node. These two components are easier to prove separately in TLA$^+$, since the proof for one side is simpler, and can later be directly reused for the other side. Here, I group right and left components of the same property in one big formula.

To relate this list of invariants to the intuitive proof discussed in Section 6.1 and make the presentation more meaningful, the invariants can be (very roughly) grouped into the following categories: basic invariants, invariants relating to the joining and probing phases, and invariants relating to coverage and stability.

Basic Invariants

These invariants refer to types and some elementary properties of the variables, the possible relationship between the values of different variables, or properties of the exchanged messages between nodes. For example, we need to show that any node that has a to-join node other than itself is a Ready node, that a node does not send duplicate messages to the same recipient, or that the sender of a lease reply message is always a Ready/OK node. This group of invariants can be seen as the beginning of the invariant food chain, being straight-forward to prove and heavily used in the proof of the more complex invariants.

Invariant 3 (Type Invariant)**.**

$\wedge\ Status \in [I \rightarrow \{\text{"Ready"}, \text{"OK"}, \text{"Waiting"}, \text{"Dead"}\}]$
$\wedge\ Leases \in [I \rightarrow \text{SUBSET } I] \wedge Grants \in [I \rightarrow \text{SUBSET } I]$
$\wedge\ MessagePool \in \text{SUBSET } Message \wedge ToJoin \in [I \rightarrow I]$
$\wedge\ RoutingTables \in [I \rightarrow RoutingTable]$
$\wedge\ LeafSets \in [I \rightarrow LeafSet] \wedge \forall\, i \in I : LeafSets[i].node = i$
$\wedge\ Probing \in [I \rightarrow \text{SUBSET } I] \wedge Failed \in [I \rightarrow \text{SUBSET } I]$

Invariant 4. *There is always at least one Ready node, and therefore at least one participating node.*

$ReadyNodes \neq \{\} \wedge ReadyOKNodes \neq \{\} \wedge ParticipatingNodes \neq \{\}$

Invariant 5. *All leaf sets are balanced, proper, and organized.*

$\forall\, i \in I : \text{LET } ls \triangleq LeafSets[i]$
$\quad\quad\quad\quad \text{IN } IsBalanced(ls) \wedge IsProper(ls) \wedge IsOrganized(ls)$

Invariant 6. *The leaf set of a participating node consists solely of other participating nodes.*

$\forall\, i \in ParticipatingNodes :$
$\quad LeafSetContent(LeafSets[i]) \subseteq ParticipatingNodes$

Invariant 7. *A Dead node is not a to-join node, and consequently, not a participating node.*

$DeadNodes \cap (ToJoinNodes \cup ParticipatingNodes) = \{\}$

Invariant 8. *The probe set of a node i consists of participating nodes.*

$\forall\, i \in I : Probing[i] \subseteq ParticipatingNodes$

Invariant 9. *A Ready/OK node i only has leases from other Ready or OK nodes. A node i that is not Ready/OK does not have leases from any other nodes.*

$\wedge \forall i \in ReadyOKNodes : Leases[i] \subseteq ReadyOKNodes$
$\wedge \forall i \in I \setminus ReadyOKNodes : Leases[i] = \{i\}$

Invariant 10. *Only Ready nodes can have to-join nodes. No two nodes can have the same to-join node.*

$\wedge \forall i \in I : ToJoin[i] \neq i \Rightarrow Status[i] = \text{"Ready"}$
$\wedge \forall i, j, k \in I : ToJoin[i] = k \wedge ToJoin[j] = k \wedge i \neq k \wedge j \neq k \Rightarrow i = j$

Invariant 11. *A node that has issued a pending join request is Waiting, not participating, and has an empty leaf set and probe set.*

$\forall m \in MessagePool : m.content.type = \text{"JoinRequest"} \Rightarrow$
$\quad \wedge Status[m.content.node] = \text{"Waiting"}$
$\quad \wedge m.content.node \notin ParticipatingNodes$
$\quad \wedge LeafSets[m.content.node] = EmptyLS(m.content.node)$
$\quad \wedge Probing[m.content.node] = \{\}$

Invariant 12. *Let m be a pending join reply message. Then m is issued by a Ready node x to a Waiting node y, where $ToJoin[x] = y$. Additionally, y is not contained in the leaf set attached to m, and has an empty leaf set and probe set.*

$\forall m \in MessagePool : m.content.type = \text{"JoinReply"} \Rightarrow$
$\quad \wedge Status[m.content.ls.node] = \text{"Ready"}$
$\quad \wedge Status[m.destination] = \text{"Waiting"}$
$\quad \wedge m.destination = ToJoin[m.content.ls.node]$
$\quad \wedge m.destination \notin LeafSetContent(m.content.ls)$
$\quad \wedge LeafSets[m.destination] = EmptyLS(m.destination)$
$\quad \wedge Probing[m.destination] = \{\}$

Invariant 13. *Nodes do not send messages to themselves.*

$$
\begin{aligned}
&\forall\, m \in MessagePool : \\
&\quad \wedge\, m.content.type \in \{\text{“Probe”}, \text{“ProbeReply”}, \text{“LeaseRequest”}\} \\
&\quad\ \Rightarrow m.content.node \neq m.destination \\
&\quad \wedge\, m.content.type \in \{\text{“JoinReply”}, \text{“ProbeReply”}, \text{“LeaseReply”}\} \\
&\quad\ \Rightarrow m.content.ls.node \neq m.destination
\end{aligned}
$$

Invariant 14. *There is at most one pending join request per node.*

$$
\begin{aligned}
&\forall\, m1,\, m2 \in MessagePool : \\
&\quad \wedge\, m1.content.type = \text{“JoinRequest”} \wedge m2.content.type = \text{“JoinRequest”} \\
&\quad \wedge\, m1 \neq m2 \\
&\quad \Rightarrow m1.content.node \neq m2.content.node
\end{aligned}
$$

Invariant 15. *There is at most one pending join reply per node.*

$$
\begin{aligned}
&\forall\, m1,\, m2 \in MessagePool : \\
&\quad \wedge\, m1.content.type = \text{“JoinReply”} \wedge m2.content.type = \text{“JoinReply”} \\
&\quad \wedge\, m1 \neq m2 \\
&\quad \Rightarrow m1.destination \neq m2.destination
\end{aligned}
$$

Invariant 16. *For any node, there cannot be a pending join request from and a pending join reply to that node at the same time.*

$$
\begin{aligned}
&\forall\, m1,\, m2 \in MessagePool : \\
&\quad m1.content.type\ = \text{“JoinReply”} \wedge m2.content.type = \text{“JoinRequest”} \\
&\quad \Rightarrow m2.content.node \neq m1.destination
\end{aligned}
$$

Invariant 17. *No two pending probe messages are identical.*

$$
\begin{aligned}
&\forall\, m1,\, m2 \in MessagePool : \\
&\quad \wedge\, m1.content.type = \text{“Probe”} \wedge m2.content.type = \text{“Probe”} \\
&\quad \wedge\, m1.content.node = m2.content.node \\
&\quad \wedge\, m1.destination = m2.destination \\
&\quad \Rightarrow m1 = m2
\end{aligned}
$$

Invariant 18. *No two pending probe reply messages are identical.*

$$
\begin{aligned}
&\forall\, m1,\, m2 \in MessagePool : \\
&\quad \wedge\, m1.content.type = \text{“ProbeReply”} \wedge m2.content.type = \text{“ProbeReply”} \\
&\quad \wedge\, m1.content.node = m2.content.node \\
&\quad \wedge\, m1.destination = m2.destination \\
&\quad \Rightarrow m1 = m2
\end{aligned}
$$

Invariant 19. *Let m be a pending message that is either a join reply, probe, probe reply, lease request or least reply. Then both the sender and recipient of m are participating nodes.*

$$
\begin{aligned}
&\forall\, m \in MessagePool : \\
&\quad \wedge\, m.content.type \in \{\text{“Probe”},\ \text{“ProbeReply”},\ \text{“LeaseRequest”}\} \\
&\qquad \Rightarrow m.content.node \in ParticipatingNodes \\
&\quad \wedge\, m.content.type \in \{\text{“JoinReply”},\ \text{“ProbeReply”},\ \text{“LeaseReply”}\} \\
&\qquad \Rightarrow m.content.ls.node \in ParticipatingNodes \\
&\quad \wedge\, m.content.type \in \{\text{“JoinReply”},\ \text{“Probe”},\ \text{“ProbeReply”}, \\
&\qquad\qquad\qquad\qquad \text{“LeaseRequest”},\ \text{“LeaseReply”}\} \\
&\qquad \Rightarrow m.destination \in ParticipatingNodes
\end{aligned}
$$

Invariant 20. *For any two nodes x and y, there cannot be a pending probe from x to y and a pending probe reply from y to x at the same time.*

$$
\begin{aligned}
&\forall\, m1,\, m2 \in MessagePool : \\
&\quad \wedge\, m1.content.type = \text{“ProbeReply”} \wedge m2.content.type = \text{“Probe”} \\
&\quad \wedge\, m1.destination = m2.content.node \\
&\quad \wedge\, m1.content.node = m2.destination \\
&\quad \Rightarrow \text{FALSE}
\end{aligned}
$$

Invariant 21. *If there is a pending join reply to x, then there cannot be a pending probe or probe reply message from x.*

$$\begin{aligned}
&\forall\, m1,\, m2 \in MessagePool : \\
&\quad \wedge\, m1.content.type = \text{“JoinReply”} \\
&\quad \wedge\, m2.content.type \in \{\text{“Probe”},\ \text{“ProbeReply”}\} \\
&\quad \Rightarrow m2.content.node \neq m1.destination
\end{aligned}$$

Invariant 22. *If there is a pending join reply to x, then there cannot be a pending probe reply message to x.*

$$\begin{aligned}
&\forall\, m1,\, m2 \in MessagePool : \\
&\quad m1.content.type = \text{“JoinReply”} \wedge m2.content.type = \text{“ProbeReply”} \\
&\quad \Rightarrow m2.destination \neq m1.destination
\end{aligned}$$

Invariant 23. *If there is a pending probe message from a node x to a node y, then y is in the probe set of x.*

$$\begin{aligned}
&\forall\, m \in MessagePool : m.content.type = \text{“Probe”} \Rightarrow \\
&\quad m.destination \in Probing[m.content.node]
\end{aligned}$$

Invariant 24. *If there is a pending probe reply message from a node x to a node y, then x is in the probe set of y.*

$$\begin{aligned}
&\forall\, m \in MessagePool : m.content.type = \text{“ProbeReply”} \Rightarrow \\
&\quad m.content.node \in Probing[m.destination]
\end{aligned}$$

Invariant 25. *The sender of a pending lease request or lease reply message is Ready or OK.*

$$\begin{aligned}
&\forall\, m \in MessagePool : \\
&\quad \wedge\, m.content.type = \text{“LeaseRequest”} \\
&\qquad \Rightarrow Status[m.content.node] \in \{\text{“OK”},\ \text{“Ready”}\} \\
&\quad \wedge\, m.content.type = \text{“LeaseReply”} \\
&\qquad \Rightarrow Status[m.content.ls.node] \in \{\text{“OK”},\ \text{“Ready”}\}
\end{aligned}$$

Invariant 26. *Let m be a lease reply message where the lease is granted, i.e. $m.content.grant = \text{TRUE}$. Then the destination of m is a Ready or OK node.*

$$\forall\, m \in MessagePool : m.content.type = \text{"LeaseReply"} \wedge m.content.grant \\ \Rightarrow Status[m.destination] \in \{\text{"Ready"}, \text{"OK"}\}$$

Invariant 27. *The failed set of any node is empty. Also, the failed set attached to any pending message is empty.*

$$\begin{aligned} &\wedge\, \forall\, i \in I : Failed[i] = \{\} \\ &\wedge\, \forall\, m \in MessagePool : \\ &\quad m.content.type \in \{\text{"Probe"}, \text{"ProbeReply"}\} \Rightarrow \\ &\qquad m.content.failed = \{\} \end{aligned}$$

Invariant 28. *Dead nodes do not send messages.*

$$\begin{aligned} &\forall\, m \in MessagePool : \\ &\quad \wedge\, m.content.type \in \{\text{"JoinRequest"}, \text{"Probe"}, \\ &\qquad\qquad \text{"ProbeReply"}, \text{"LeaseRequest"}\} \\ &\qquad \Rightarrow Status[m.content.node] \neq \text{"Dead"} \\ &\quad \wedge\, m.content.type \in \{\text{"JoinReply"}, \text{"LeaseReply"}\} \\ &\qquad \Rightarrow Status[m.content.ls.node] \neq \text{"Dead"} \end{aligned}$$

Invariant 29. *Dead nodes have empty leaf sets and probe sets. If a Ready node has an empty leaf set, it is the only Ready node on the network.*

$$\begin{aligned} &\wedge\, \forall\, i \in DeadNodes : LeafSets[i] = EmptyLS(i) \wedge Probing[i] = \{\} \\ &\wedge\, \forall\, i \in ReadyNodes : LeafSets[i] = EmptyLS(i) \Rightarrow ReadyNodes = \{i\} \end{aligned}$$

Invariant 30. *A Waiting node that has issued a join request and has not yet received a join reply has an empty leaf set.*

$$
\begin{aligned}
&\forall\, m \in MessagePool : \\
&\quad \land\, m.content.type = \text{“JoinRequest”} \\
&\qquad\qquad \Rightarrow LeafSets[m.content.node] = EmptyLS(m.content.node) \\
&\quad \land\, m.content.type = \text{“JoinReply”} \\
&\qquad\qquad \Rightarrow LeafSets[m.destination] = EmptyLS(m.destination)
\end{aligned}
$$

Invariant 31. *A node that sends or receives probes does not have an empty leaf set.*

$$
\begin{aligned}
&\land\, \forall\, i \in I : Probing[i] \neq \{\} \Rightarrow LeafSets[i] \neq EmptyLS(i) \\
&\land\, \forall\, m \in MessagePool : \\
&\qquad \land\, m.content.type = \text{“ProbeReply”} \\
&\qquad\qquad \Rightarrow LeafSets[m.content.node] \neq EmptyLS(m.content.node) \\
&\qquad \land\, m.content.type = \text{“ProbeReply”} \\
&\qquad\qquad \Rightarrow LeafSets[m.destination] \neq EmptyLS(m.destination)
\end{aligned}
$$

Invariant 32. *Leaf set objects attached in messages are balanced, proper and organized, and comprised solely of participating nodes.*

$$
\begin{aligned}
&\forall\, m \in MessagePool : \\
&\quad m.content.type \in \{\text{“JoinReply”},\ \text{“ProbeReply”},\ \text{“LeaseReply”}\} \Rightarrow \\
&\qquad \land\, IsBalanced(m.content.ls) \\
&\qquad \land\, IsProper(m.content.ls) \\
&\qquad \land\, IsOrganized(m.content.ls) \\
&\qquad \land\, LeafSetContent(m.content.ls) \subseteq ParticipatingNodes
\end{aligned}
$$

Invariant 33. *The Waiting to-join node of a Ready node r is in the leaf set of r, and is the closest participating node to it on the ring from one side.*

$$
\begin{aligned}
&\forall\, r \in ReadyNodes : ToJoin[r] \notin ReadyOKNodes \Rightarrow \\
&\quad \land\, ToJoin[r] \in LeafSetContent(LeafSets[r]) \\
&\quad \land\, \lor\, \forall\, i \in (ParticipatingNodes \setminus \{r\}) : ClockwiseArc(r,\ ToJoin[r],\ i) \\
&\qquad \lor\, \forall\, i \in (ParticipatingNodes \setminus \{r\}) : ClockwiseArc(i,\ ToJoin[r],\ r)
\end{aligned}
$$

Invariant 34. *If there is a pending probe reply message from node i to node j, then either j is a member i's leaf set, or i's leaf set is full with closer members than j.*

$$
\begin{aligned}
&\forall\, m \in MessagePool : \\
&\quad m.content.type = \text{“ProbeReply”} \Rightarrow \\
&\qquad \vee\ m.destination \in LeafSetContent(LeafSets[m.content.node]) \\
&\qquad \vee\ \wedge\ IsComplete(LeafSets[m.content.node]) \\
&\qquad\quad \wedge\ \forall\, x \in LeafSets[m.content.node].right : \\
&\qquad\qquad\qquad ClockwiseArc(m.content.node,\ x,\ m.destination) \\
&\qquad\quad \wedge\ \forall\, x \in LeafSets[m.content.node].left : \\
&\qquad\qquad\qquad ClockwiseArc(m.destination,\ x,\ m.content.node)
\end{aligned}
$$

The Join-Probe Relation

These invariants are concerned with the relationship between a joining node and its neighbors, which is necessary to prove the stability of this joining node once it has finished joining. In particular, the aim of these invariants is to show that a Waiting node i does not turn OK before probing node j, its Ready/OK neighbor at the time it turns OK. This is done by analyzing the different possible orderings of events that led up to i and j being both Ready/OK, in a manner that corresponds to the discussion in Section 6.1.

These invariants represent, by far, the most challenging part of the proof, being the most difficult to come up with and to prove.

Invariant 35. *Let i and j be two participating nodes, where j is a joining node that has not yet received its join reply message. If j is a member of i's leaf set, then either j is joining through i, or i is probing j.*

$$
\begin{aligned}
&\forall\, i, j \in ParticipatingNodes\ ,\ m \in MessagePool : \\
&\quad \wedge\ i \neq j \wedge j \in LeafSetContent(LeafSets[i]) \\
&\quad \wedge\ m.content.type = \text{“JoinReply”} \wedge m.destination = j \\
&\Rightarrow j \in Probing[i] \vee j = ToJoin[i]
\end{aligned}
$$

Invariant 36. *Let m be a pending join reply message from r to i. Then the leaf set attached to m contains the Ready/OK neighbors of both r and i. Moreover, any node j in the leaf set attached to m that lies between i and its Ready/OK neighbor is the to-join node of that neighbor.*

$$
\begin{array}{l}
\forall\, m \in MessagePool : \\
\quad m.content.type = \text{“}\mathsf{JoinReply}\text{”} \Rightarrow \\
\qquad \text{LET } CR1 \triangleq ClosestFromTheRight(m.destination, \\
\qquad\qquad\qquad\qquad ReadyOKNodes) \\
\qquad\qquad CL1 \triangleq ClosestFromTheLeft(m.destination, \\
\qquad\qquad\qquad\qquad ReadyOKNodes) \\
\qquad\qquad CR2 \triangleq ClosestFromTheRight(m.content.ls.node, \\
\qquad\qquad\qquad\qquad ReadyOKNodes \setminus \{m.content.ls.node\}) \\
\qquad\qquad CL2 \triangleq ClosestFromTheLeft(m.content.ls.node, \\
\qquad\qquad\qquad\qquad ReadyOKNodes \setminus \{m.content.ls.node\}) \\
\qquad \text{IN} \\
\qquad\quad \wedge\, CR1 \in LeafSetContent(m.content.ls) \\
\qquad\quad \wedge\, CL1 \in LeafSetContent(m.content.ls) \\
\qquad\quad \wedge\, CR2 \in LeafSetContent(m.content.ls) \\
\qquad\quad \wedge\, CL2 \in LeafSetContent(m.content.ls) \\
\qquad\quad \wedge\, \forall\, j \in LeafSetContent(m.content.ls) : \\
\qquad\qquad\quad \wedge\, j \neq CR1 \\
\qquad\qquad\quad \wedge\, ClockwiseArc(m.destination,\, j,\, CR1) \\
\qquad\qquad\quad \Rightarrow j = ToJoin[CR1] \\
\qquad\quad \wedge\, \forall\, j \in LeafSetContent(m.content.ls) : \\
\qquad\qquad\quad \wedge\, j \neq CL1 \\
\qquad\qquad\quad \wedge\, ClockwiseArc(CL1,\, j,\, m.destination) \\
\qquad\qquad\quad \Rightarrow j = ToJoin[CL1]
\end{array}
$$

Invariant 37. *Let i be a non-Ready/OK node that is joining through some Ready node r. Let CR and CL be r's right and left Ready/OK neighbors. If another non-Ready/OK k is currently joining through CR (or CL), then one of the following must hold: (1) The leaf set of i is empty; that is, i has not yet received its join reply or begun the probing phase. (2) the leaf set of k is empty; that is, k has not yet received its join reply or begun the probing phase, (3) i is probing k, (4) k is probing i, (5) k is probing r, (6) k is a member of i's leaf set.*

$$
\begin{aligned}
&\forall\, i \in ParticipatingNodes \setminus ReadyOKNodes,\ r \in ReadyNodes : \\
&\quad \forall\, k \in \{ClosestFromTheRight(r,\ ReadyOKNodes \setminus \{r\}), \\
&\qquad\qquad ClosestFromTheLeft(r,\ ReadyOKNodes \setminus \{r\})\} \\
&\qquad \wedge\ i = ToJoin[r] \wedge ToJoin[k] \neq k \wedge ToJoin[k] \neq i \\
&\qquad \wedge\ ToJoin[k] \notin ReadyOKNodes \\
&\qquad \Rightarrow\ \vee\ LeafSets[ToJoin[k]] = EmptyLS(ToJoin[k]) \\
&\qquad\quad\ \vee\ LeafSets[i] = EmptyLS(i) \vee ToJoin[k] \in Probing[i] \\
&\qquad\quad\ \vee\ i \in Probing[ToJoin[k]] \vee r \in Probing[ToJoin[k]] \\
&\qquad\quad\ \vee\ ToJoin[k] \in LeafSetContent(LeafSets[i])
\end{aligned}
$$

Invariant 38. *Suppose there is a pending join reply message to some node x, and a pending probe reply message to some node y, both messages from the same node r (note that r must be a Ready node by Invariant 12). Then one of the following must hold: (1) x is in the leaf set attached to the message to y, (2) y is in the leaf set attached to the message to x, (3) y is not in r's leaf set, and the leaf set of r is full with closer nodes to it than y.*

$$
\begin{aligned}
&\forall\, m1,\ m2 \in MessagePool : \\
&\quad \wedge\ m1.content.type = \text{“JoinReply”} \\
&\quad \wedge\ m2.content.type = \text{“ProbeReply”} \\
&\quad \wedge\ m1.content.ls.node = m2.content.node \\
&\Rightarrow\ \vee\ m1.destination \in LeafSetContent(m2.content.ls) \\
&\qquad \vee\ m2.destination \in LeafSetContent(m1.content.ls) \\
&\qquad \vee\ \wedge\ IsComplete(LeafSets[m2.content.node]) \\
&\qquad\quad \wedge\ m2.destination \notin LeafSetContent(LeafSets[m2.content.node]) \\
&\qquad\quad \wedge\ \forall\, x \in LeafSets[m2.content.node].right : \\
&\qquad\qquad\quad ClockwiseArc(m2.content.node,\ x,\ m2.destination) \\
&\qquad\quad \wedge\ \forall\, x \in LeafSets[m2.content.node].left : \\
&\qquad\qquad\quad ClockwiseArc(m2.destination,\ x,\ m2.content.node)
\end{aligned}
$$

Invariant 39. *Let i be a Ready/OK node, and k be a non-Ready/OK node joining through i's right or left Ready neighbor r. If k is not a member of i's leaf set, then either k's leaf set is empty or k is probing i.*

$$
\begin{array}{l}
\forall\, i \in ReadyOKNodes : \\
\quad \forall\, r \in \{ClosestFromTheRight(i,\, ReadyOKNodes \setminus \{i\}) \\
\qquad\qquad ClosestFromTheLeft(i,\, ReadyOKNodes \setminus \{i\})\} : \\
\qquad ToJoin[r] \notin (LeafSetContent(LeafSets[i]) \cup ReadyOKNodes) \Rightarrow \\
\qquad\quad LeafSets[ToJoin[r]] = EmptyLS(ToJoin[r]) \vee i \in Probing[ToJoin[r]]
\end{array}
$$

Invariant 40. *Let m be a pending join reply from r to i. Let n be i's right or left participating neighbor. Then if n is not Ready/OK, one of the following must hold: (1) n is in the leaf set attached to m, (2) n's leaf set is empty; that is, n has not yet received its own join reply or begun the probing phase, (3) n is probing i, (4) n is probing r.*

$$
\begin{array}{l}
\forall\, m \in MessagePool : \\
\quad \text{LET } r \triangleq m.content.ls.node \\
\qquad\quad i \triangleq m.destination \\
\quad \text{IN} \quad \forall\, n \in \{ClosestFromTheRight(r,\, ParticipatingNodes \setminus \{r\}), \\
\qquad\qquad\qquad ClosestFromTheLeft(r,\, ParticipatingNodes \setminus \{r\})\} \\
\qquad\quad m.content.type = \text{“JoinReply”} \wedge n \notin ReadyOKNodes \Rightarrow \\
\qquad\qquad \vee\, n \in LeafSetContent(m.content.ls) \vee LeafSets[n] = EmptyLS(n) \\
\qquad\qquad \vee\, i \in Probing[n] \vee r \in Probing[n]
\end{array}
$$

Invariant 41. *Let i be a non-Ready/OK node joining through a Ready node r. Let k be another non-Ready/OK node joining through one of r's Ready neighbors n (here the right or left Ready/OK neighbor). If i is a member of k's leaf set, then one of the following must hold: (1) i is probing k, (2) k is probing i, (3) k is a member of i's leaf set.*

$$
\begin{array}{l}
\forall\, i \in ParticipatingNodes \setminus ReadyOKNodes,\, r \in ReadyNodes : \\
\quad \forall\, n \in \{ClosestFromTheRight(r,\, ReadyOKNodes \setminus \{r\}), \\
\qquad\qquad ClosestFromTheLeft(r,\, ReadyOKNodes \setminus \{r\})\} \\
\qquad \wedge\, ToJoin[n] \neq n \wedge ToJoin[n] \neq i \\
\qquad \wedge\, ToJoin[n] \notin ReadyOKNodes \wedge i = ToJoin[r] \\
\qquad \wedge\, i \in LeafSetContent(LeafSets[ToJoin[n]]) \\
\qquad\ \Rightarrow\ \vee\, ToJoin[n] \in Probing[i] \vee i \in Probing[ToJoin[n]] \\
\qquad\qquad \vee\, ToJoin[n] \in LeafSetContent(LeafSets[i])
\end{array}
$$

Invariant 42. *Let i be a non-Ready/OK node joining through a Ready node r. Let k be a non-Ready/OK node joining through one of r's Ready neighbors n (here the right or left Ready/OK neighbor of r). If there is a pending probe reply message m from r to k, then one of the following must hold: (1) i's leaf set is empty, (2) i is in the leaf set attached to m, (3) i is probing k, (4) k is a member of i's leaf set.*

$$
\begin{aligned}
&\forall\, i \in ParticipatingNodes \setminus ReadyOKNodes,\ r \in ReadyNodes,\\
&\quad m \in MessagePool :\\
&\quad \forall\, n \in \{ClosestFromTheRight(r,\ ReadyOKNodes \setminus \{r\}),\\
&\qquad\qquad ClosestFromTheLeft(r,\ ReadyOKNodes \setminus \{r\})\}\\
&\qquad \wedge\, i = ToJoin[r] \wedge m.content.type = \text{“ProbeReply”}\\
&\qquad \wedge\, m.content.node = r \wedge m.destination = ToJoin[n]\\
&\qquad \wedge\, ToJoin[n] \neq n \wedge ToJoin[n] \neq i\\
&\qquad \wedge\, ToJoin[n] \notin ReadyOKNodes \wedge LeafSets[i] \neq EmptyLS(i)\\
&\qquad \Rightarrow\ \vee\, i \in LeafSetContent(m.content.ls)\\
&\qquad\qquad \vee\, m.destination \in Probing[i]\\
&\qquad\qquad \vee\, m.destination \in LeafSetContent(LeafSets[i])
\end{aligned}
$$

Invariant 43. *Let m be a pending join reply message to some node i from some node j. Let k be another non-Ready/OK node joining through one of j's Ready neighbors r. Then one of the following must hold: (1) k is in the leaf set attached to m, (2) k's leaf set is empty; that is k has not yet received its join reply or begun the probing phase, (3) k is probing i, (4) k is probing r.*

$$
\begin{aligned}
&\forall\, m \in MessagePool :\\
&\quad \forall\, r \in \{ClosestFromTheRight(m.content.ls.node,\\
&\qquad\qquad ReadyOKNodes \setminus \{m.content.ls.node\}),\\
&\qquad\quad ClosestFromTheLeft(m.content.ls.node,\\
&\qquad\qquad ReadyOKNodes \setminus \{m.content.ls.node\})\} :\\
&\qquad \wedge\, m.content.type = \text{“JoinReply”} \wedge m.destination \neq ToJoin[r]\\
&\qquad \wedge\, r \neq ToJoin[r] \wedge ToJoin[r] \notin ReadyOKNodes\\
&\qquad \Rightarrow\ \vee\, ToJoin[r] \in LeafSetContent(m.content.ls)\\
&\qquad\qquad \vee\, LeafSets[ToJoin[r]] = EmptyLS(ToJoin[r])\\
&\qquad\qquad \vee\, m.destination \in Probing[ToJoin[r]]\\
&\qquad\qquad \vee\, m.content.ls.node \in Probing[ToJoin[r]]
\end{aligned}
$$

Invariant 44. *Let i be a non-Ready/OK node and r its left or right Ready/OK neighbor. Then either i's leaf set is empty; that is i has not yet received its join reply message or begun the probing phase, r is a member of i's leaf set, or i is probing r.*

$$
\begin{aligned}
&\forall\, i \in \mathit{ParticipatingNodes} \setminus \mathit{ReadyOKNodes} : \\
&\quad \forall\, r \in \{\mathit{ClosestFromTheRight}(i,\ \mathit{ReadyOKNodes}), \\
&\qquad\qquad \mathit{ClosestFromTheLeft}(i,\ \mathit{ReadyOKNodes})\} : \\
&\qquad \vee\ \mathit{LeafSets}[i] = \mathit{EmptyLS}(i) \vee r \in \mathit{LeafSetContent}(\mathit{LeafSets}[i]) \\
&\qquad \vee\ r \in \mathit{Probing}[i]
\end{aligned}
$$

Invariant 45. *Let i be a non-Ready/OK node and r its left or right Ready/OK neighbor. Then either i's leaf set is empty; that is i has not yet received its join reply message or begun the probing phase, i is a member of r's leaf set, or i is probing r.*

$$
\begin{aligned}
&\forall\, i \in \mathit{ParticipatingNodes} \setminus \mathit{ReadyOKNodes} : \\
&\quad \forall\, r \in \{\mathit{ClosestFromTheRight}(i,\ \mathit{ReadyOKNodes}), \\
&\qquad\qquad \mathit{ClosestFromTheLeft}(i,\ \mathit{ReadyOKNodes})\} : \\
&\qquad \vee\ \mathit{LeafSets}[i] = \mathit{EmptyLS}(i) \vee r \in \mathit{Probing}[i] \\
&\qquad \vee\ i \in \mathit{LeafSetContent}(\mathit{LeafSets}[r])
\end{aligned}
$$

Coverage and Stability

Finally, the following invariants relate to exclusive coverage between different Ready/OK nodes and are the main invariants used to prove correct delivery. The proof of these invariants and the join/probe invariants of the previous section are heavily inter-dependent. The difficulty in proving these invariants lies in the rather involved arithmetic reasoning on coverage.

Invariant 46. *A Ready node i does not cover a non-Ready/OK k that is the to-join node of (1) itself, (2) its leaf set neighbor, (3) any node j that has granted i a lease, and (4) any node j that i has probed and from which there is a pending probe reply message to i.*

$$
\begin{aligned}
&\forall\, i, j \in \mathit{ReadyNodes} : \\
&\quad \land \lor j = i \lor j \in \mathit{Leases}[i] \\
&\qquad \lor j = \mathit{LeftNeighbor}(\mathit{LeafSets}[i]) \lor j = \mathit{RightNeighbor}(\mathit{LeafSets}[i]) \\
&\qquad \lor \exists\, m \in \mathit{MessagePool} : \\
&\qquad\quad \land m.\mathit{content.type} = \text{“ProbeReply”} \\
&\qquad\quad \land m.\mathit{content.node} = j \land m.\mathit{destination} = i \\
&\quad \land \mathit{ToJoin}[j] \notin \mathit{ReadyOKNodes} \\
&\quad \Rightarrow \neg \mathit{Covers}(\mathit{LeafSets}[i],\ \mathit{ToJoin}[j])
\end{aligned}
$$

Invariant 47. *A Ready node r does not cover any to-join node i.*

$$
\forall\, i \in \mathit{ToJoinNodes},\ r \in \mathit{ReadyNodes} : r \neq i \Rightarrow \neg \mathit{Covers}(\mathit{LeafSets}[r],\ i)
$$

Invariant 48 (Exclusive Coverage)**.** *Two Ready/OK nodes i and j do not both think they cover the same key k.*

$$
\begin{aligned}
&\forall\, i, j \in \mathit{ReadyOKNodes},\ k \in I : \\
&\quad i \neq j \land i \neq k \land j \neq k \\
&\qquad \Rightarrow \neg \mathit{Covers}(\mathit{LeafSets}[i],\ k) \lor \neg \mathit{Covers}(\mathit{LeafSets}[j],\ k)
\end{aligned}
$$

Invariant 49 (Stable Network)**.** *Let i be a Ready/OK node and r its left or right Ready/OK neighbor. Then r is a member of i's leaf set.*

$$
\begin{aligned}
&\mathit{StableNetwork} \triangleq \\
&\quad \forall\, i \in \mathit{ReadyOKNodes} : \\
&\qquad \land \mathit{ClosestFromTheRight}(i,\ \mathit{ReadyOKNodes} \setminus \{i\}) \\
&\qquad\quad \in \mathit{LeafSetContent}(\mathit{LeafSets}[i]) \\
&\qquad \land \mathit{ClosestFromTheLeft}(i,\ \mathit{ReadyOKNodes} \setminus \{i\}) \\
&\qquad\quad \in \mathit{LeafSetContent}(\mathit{LeafSets}[i])
\end{aligned}
$$

6.4 An Outline of the Reduction Proof

In what follows, I show that strong correct delivery can be derived from the correctness invariants presented above.

Lemma 1. *A proper, balanced and organized leaf set does not cover its own members.*

$\forall\, ls \in LeafSet : \forall\, i \in (ls.left \cup ls.right) :$
$\quad IsProper(ls) \wedge IsBalanced(ls) \wedge IsOrganized(ls) \Rightarrow \neg Covers(ls,\, i)$

Proof. By definition of coverage and the leaf set properties "proper", "balanced" and "organized". □

Lemma 2. *In a stable network where the leaf set invariants hold, the leaf set of a Ready/OK node i does not cover any other Ready/OK node j.*

$StableNetwork \wedge LeafSetInvariants \Rightarrow$
$\quad \forall\, i, j \in ReadyOKNodes : i \neq j \Rightarrow \neg Covers(LeafSets[i],\, j)$

Proof. For the sake of contradiction, let i and j be Ready/OK nodes where $i \neq j$ and i's leaf set covers j. Let l, r be the closest Ready/OK nodes to i from the left and right, respectively. By Lemma 1, j is not a member of i's leaf set. Because the network is stable, this means that $j \neq l$ and $j \neq r$. This means that j is separated from i by leaf set members l and r on both sides. □

Theorem 1. *Correct delivery follows from the correctness invariants listed in Section 6.3.*

$CorrectnessInvariants \Rightarrow StrongCorrectDelivery$

Proof. Recall the definition of *StrongCorrectDelivery* (Invariant 2). Let i and k be two different Ready nodes and assume i covers some key j. Let m be a pending lookup message for key j addressed to i. It is required to show that

(1) i is closer to j than k is in terms of absolute ring distance, and (2) k does not cover j.

$$\wedge\ \mathit{AbsoluteDistance}(i,\, j) \leq \mathit{AbsoluteDistance}(k,\, j)$$
$$\wedge\ \neg \mathit{Covers}(\mathit{LeafSets}[k],\, j)$$

For brevity, let us denote $\mathit{AbsoluteDistance}(i, j)$ and $\mathit{AbsoluteDistance}(k, j)$ by D_i and D_k, respectively. The leaf sets of i and k are denoted by ls_i and ls_k. The right and left leaf set neighbors of i are RN_i and LN_i, respectively, and similarly for k, we use RN_k and LN_k. The right and left coverage bounds of i's leaf set are denoted by RC_i and LC_i (RC_k and LC_k for k).

We can safely assume that $i \neq j$, since for $i = j$ it is trivial to conclude that (1) i is closer to i than k is (the distance between a node and itself is zero), and (2) k does not cover i, by Lemma 2. Similarly, we also know that $j \neq k$, since i covers j but cannot cover k, also by Lemma 2.

That ls_k does not cover j now follows directly from Invariant 48, which says that no two Ready/OK nodes can cover the same key. It is left to show that $D_i \leq D_k$.

Without loss of generality, we can assume that j is right-covered by i (the proof works similarly for the left side). The situation is shown in Figure 6.4. Note that k and RN_i may be the same node, but the figure shows the general case where k is a node after RN_i on the clockwise direction from i.

By definition of coverage, the clockwise distance from i to j is no more than half the distance from i to its right neighbor RN_i,

$$\mathit{ClockwiseDistance}(i, j) \leq \mathit{ClockwiseDistance}(i, RN_i) \div 2.$$

From this, it can also be deduced,

$$\mathit{ClockwiseDistance}(i, j) \leq \mathit{ClockwiseDistance}(j, RN_i)$$
$$\mathit{ClockwiseDistance}(i, j) \leq \mathit{RingSize} \div 2$$

Therefore, it must be that $D_i = \mathit{ClockwiseDistance}(i, j)$. That is, the shortest distance between i and j is the clockwise distance from i to j.

The absolute/shortest distance D_k between j and k can be either the clockwise distance from k to j, or the clockwise distance from j to k. Assume that $D_k = \mathit{ClockwiseDistance}(k, j)$. This means that the shortest path between k and j passes through i. In this case,

$$D_k = \mathit{ClockwiseDistance}(k, i) + D_i$$
$$D_k > D_i$$

On the other hand, if $D_k = \mathit{ClockwiseDistance}(j, k)$, then,

$$\begin{aligned}
D_k &= \mathit{ClockwiseDistance}(j, k) \\
&= \mathit{ClockwiseDistance}(j, RN_i) + \mathit{ClockwiseDistance}(RN_i, k) \\
&\geq \mathit{ClockwiseDistance}(i, j) + \mathit{ClockwiseDistance}(RN_i, k) \\
&\geq \mathit{ClockwiseDistance}(i, j) \\
D_k &\geq D_i
\end{aligned}$$

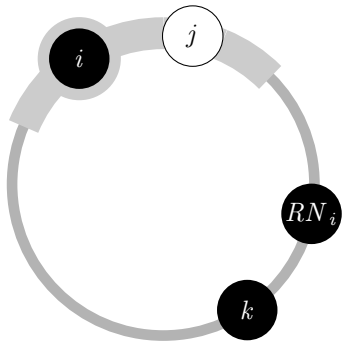


Figure 6.4 – Possible arrangement for Ready/OK nodes i and k with respect to a key j right-covered by i.

In both cases, we reach the conclusion that $D_i \leq D_k$. □

With the TLA$^+$ proof summarized in this chapter, I establish the following. Given a leaf set size of at least three, the LuPastry$^+$ network is always stable; that is, every Ready/OK node always knows its true left and right Ready/OK neighbors on the ring. As a consequence, the coverage regions of Ready/OK nodes do not overlap. It is therefore guaranteed that for any lookup request for some key k, at most one Ready node i receives this request, and this node i is (1) the node with the closest ID to key k in terms of shortest distance on the ring, and (2) the only Ready node that covers key k.

In the next chapter, I present a simplified join process for LuPastry$^+$, and show how the proof presented here can be adapted to prove correct delivery for the simplified version of the protocol.

Chapter 7

Simplified LuPastry$^+$

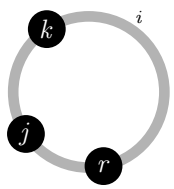
By examining the proof of correctness of LuPastry$^+$ presented in Chapter 6, it becomes clear that the lease exchange phase of the join process does not play a role in preserving the invariance of correct delivery. In this chapter, I show that this lease exchange phase is, in fact, unnecessary for achieving correct delivery and can be omitted. I present a modified version of LuPastry$^+$, which I denote by Simplified LuPastry$^+$, where a Waiting node directly turns Ready after finishing the probing process, without exchanging leases with its leaf set neighbors. I also show how the TLA$^+$ proof of correct delivery in Chapter 6 is adapted for the simplified specification, thereby obtaining a new, complete proof of correct delivery for Simplified LuPastry$^+$.

7.1 A Simplified Join Process for LuPastry$^+$

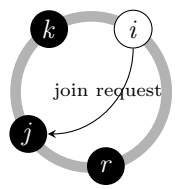
The main difference in Simplified LuPastry$^+$ to the original specification of LuPastry$^+$ is a simpler join process, shown in Figure 7.1. Compare this figure to the original join process in Figure 4.3.

The simplified process starts in the same manner as the original one–a Dead node i that wishes to join the network issues a join request, which is eventually forwarded to the Ready node k that covers key i. If k is not currently facilitating the join of any other node, it responds to i's join request, attaching its own leaf set. Node i receives k's reply and begins the probing phase.

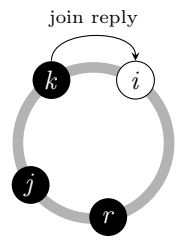
The key difference is introduced at the point when i has finished the probing phase. In Simplified LuPastry$^+$, as soon as i has finished the probing phase, it directly changes its status to Ready, instead of OK. At this point, i has officially finished the join process; it can directly help new nodes join the network. It does not need to send lease requests to its leaf set neighbors, but instead, sends a notification message to the responsible node k, letting it know that it has finished joining. Once node k has received i's notification, it may help other new nodes join the network.



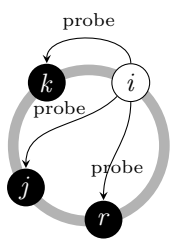
(a) Dead node i wants to join the network.



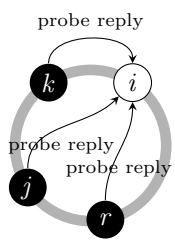
(b) Node i turns "Waiting" and sends a join request to node j.



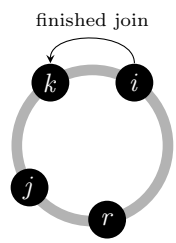
(c) The join request is forwarded to node k, and k responds to i.



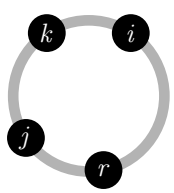
(d) Node i receives node k's join reply and begins the probing phase.



(e) Live nodes respond to i's probe messages.



(f) Node i turns Ready and notifies k. Node i may now accept join requests.



(g) Node k receives i's notification; k may accept new join requests.

Figure 7.1 – The Simplified LuPastry+ join protocol.

7.2 The TLA$^+$ Specification of Simplified LuPastry$^+$

I adapt the specification of LuPastry$^+$ explained in Section 4.2 as follows. The specification constants, ring description and distance operators remain unchanged in Simplified LuPastry$^+$. Similarly, the data structures for the routing table and leaf set are identical in both specification. The only change to the specification data structures relates to the types of messages exchanged between nodes. Lookup, joining and probing messages work as before. Lease request and reply messages are no longer needed and therefore eliminated from the specification. A new type of message is needed for the final notification at the end of the join process.

Definition 36 (Simplified LuPastry$^+$ Message)**.**

$$
\begin{array}{lcl}
msg\_Lookup & \triangleq & [type : \{\text{"Lookup"}\},\ node : I] \\
msg\_NoLegalRoute & \triangleq & [type : \{\text{"NoLegalRoute"}\},\ \ key : I] \\
msg\_JoinRequest & \triangleq & [type : \{\text{"JoinRequest"}\}, \\
 & & \quad rt : RoutingTable,\ node : I] \\
msg\_JoinReply & \triangleq & [type : \{\text{"JoinReply"}\}, \\
 & & \quad rt : RoutingTable,\ ls : LeafSet] \\
msg\_Probe & \triangleq & [type : \{\text{"Probe"}\},\ node : I, \\
 & & \quad ls : LeafSet,\ failed : \text{SUBSET } I] \\
msg\_ProbeReply & \triangleq & [type : \{\text{"ProbeReply"}\},\ node : I, \\
 & & \quad ls : LeafSet,\ failed : \text{SUBSET } I] \\
msg\_Notify & \triangleq & [type : \{\text{"Notify"}\},\ node : I]
\end{array}
$$

$$
\begin{array}{lcl}
MessageContent & \triangleq & \\
\quad msg\_Lookup \cup msg\_NoLegalRoute \cup msg\_JoinRequest \cup msg\_JoinReply & & \\
\quad \cup\ msg\_Probe \cup msg\_ProbeReply \cup msg\_Notify & & \\
 & & \\
Message & \triangleq & [destination : I,\ content : MessageContent]
\end{array}
$$

The variables of the new specification are given by the following definition.

Definition 37 (Simplified LuPastry$^+$ Variables)**.**

$$
\begin{array}{lcl}
vars & \triangleq & \langle MessagePool,\ Status,\ LeafSets,\ RoutingTables,\ Probing, \\
 & & \quad ToJoin,\ Responsible,\ Failed\rangle
\end{array}
$$

Compared to the state variables of LuPastry$^+$ in Definition 14, the original LuPastry$^+$ variables *Leases* and *Grants* were omitted, since nodes do not exchange leases. Instead, a new variable *Responsible* is added, where $Responsible[i]$ is the Ready node, if any, that is currently helping or has

helped i join the network.[1] If i is a non-participating or an initially Ready node, $Responsible[i] = i$. There is a need for this new variable in Simplified LuPastry$^+$ so that a joining node i notifies its responsible node as soon as it has turned Ready. In the original LuPastry$^+$, this notification was done implicitly through the process of lease exchange: the responsible node k for i is always one of i's leaf set neighbors, and by exchanging leases with i, k is notified that i has finished its join process.

The new type invariant is given by the following formula.

Invariant 50 (Simplified LuPastry$^+$ Type Invariant)**.**

$$
\begin{array}{l}
\land\ MessagePool \in \text{SUBSET } Message \\
\land\ Status \in [I \rightarrow \{\text{"Ready"}, \text{"Waiting"}, \text{"Dead"}\}] \\
\land\ RoutingTables \in [I \rightarrow RoutingTable] \\
\land\ LeafSets \in [I \rightarrow LeafSet] \quad \land \forall\, i \in I : LeafSets[i].node = i \\
\land\ Probing \in [I \rightarrow \text{SUBSET } I] \land Failed \in [I \rightarrow \text{SUBSET } I] \\
\land\ ToJoin \in [I \rightarrow I] \land Responsible \in [I \rightarrow I]
\end{array}
$$

Because nodes never have the status "OK" in Simplified LuPastry$^+$, it is always the case that $OKNodes = \{\}$, and therefore $ReadyOKNodes = ReadyNodes$. In the TLA$^+$ files, I simply omit the definitions of $OKNodes$ and $ReadyOKNodes$, and replace all references to $ReadyOKNodes$ with $ReadyNodes$. For the scope of this chapter, $ReadyOKNodes$ and $ReadyNodes$ are interchangeable.

The initial state of Simplified LuPastry$^+$ is modified from the original Definition 15 to accommodate the change in variables.

Definition 38 (Simplified LuPastry$^+$ Initial State)**.**

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ MessagePool = \{\} \\
\quad \land\ Status = [i \in I \mapsto \text{IF } i \in A \text{ THEN } \text{"Ready"} \text{ ELSE } \text{"Dead"}] \\
\quad \land\ ToJoin = [i \in I \mapsto i] \\
\quad \land\ Responsible = [i \in I \mapsto i] \\
\quad \land\ Probing = [i \in I \mapsto \{\}] \\
\quad \land\ Failed = [i \in I \mapsto \{\}] \\
\quad \land\ LeafSets = [i \in I \mapsto \text{IF } i \in A \text{ THEN } AddToLS(A, EmptyLS(i)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } EmptyLS(i)] \\
\quad \land\ RoutingTables = [i \in I \mapsto \text{IF } i \in A \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } AddToRT(A, EmptyRT, i) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } AddToRT(\{i\}, EmptyRT, i)]
\end{array}
$$

[1] Since each node joins only once, there is no need to reset $Responsible[i]$ to i after it has finished joining.

The next-state relation in Simplified LuPastry$^+$ is defined as follows.

Definition 39 (Simplified LuPastry$^+$ Next-State Relation)**.**

$$
\begin{array}{l}
Next \triangleq \\
\quad \exists\, i, j \in I : \\
\qquad \lor\ Lookup(i, j) \\
\qquad \lor\ RouteLookup(i, j) \\
\qquad \lor\ DeliverLookup(i, j) \\
\qquad \lor\ Join(i, j) \\
\qquad \lor\ RouteJoinRequest(i, j) \\
\qquad \lor\ ReceiveJoinRequest(i) \\
\qquad \lor\ ReceiveJoinReply(i) \\
\qquad \lor\ ReceiveProbe(i) \\
\qquad \lor\ ReceiveProbeReply(i) \\
\qquad \lor\ ReceiveNotification(i) \\
\qquad \lor\ LoseMessage
\end{array}
$$

The actions *Lookup*, *RouteLookup*, *DeliverLookup*, *Join*, *RouteJoinRequest*, *ReceiveJoinRequest*, *ReceiveProbe* and *LoseMessage* had no effect on the variables *Leases* and *Grants* in the original specification. Therefore, they are defined in the same way in Simplified LuPastry$^+$, aside from replacing the post-conditions UNCHANGED *Leases* and UNCHANGED *Grants* by the post-condition UNCHANGED *Responsible*.

The action *ReceiveJoinReply* also had no effect on the variables *Leases* and *Grants* in the original specification. However, on receiving a join reply, a node must set its *Responsible* variable to the sender of the join reply message. Therefore, in the new action formula, the post-conditions UNCHANGED *Leases* and UNCHANGED *Grants* are removed, and the variable *Responsible* is updated as follows.

Definition 40 (Simplified LuPastry$^+$ Action Receive Join Reply)**.**

$$
\begin{array}{l}
ReceiveJoinReply(i) \triangleq \\
\quad Status[i] = \text{"Waiting"} \land \exists\, m \in MessagePool : \\
\qquad \land\, m.content.type = \text{"JoinReply"} \land m.destination = i \\
\qquad \land\, \text{LET}\ ls \triangleq m.content.ls \\
\qquad\qquad\quad lsc \triangleq LeafSetContent(ls) \\
\qquad\qquad\quad nrt \triangleq AddToRT(lsc \cup RTContent(m.content.rt), \\
\qquad\qquad\qquad\qquad\qquad\qquad RoutingTables[i],\ i) \\
\qquad\qquad\quad nls \triangleq AddToLS(lsc,\ LeafSets[i]) \\
\qquad\qquad\quad prb \triangleq LeafSetContent(nls) \setminus \{i\} \\
\qquad\qquad\quad msg \triangleq ProbeSet(i,\ nls,\ \{\},\ prb) \\
\qquad\quad \text{IN}\ \ \land\, RoutingTables' = [RoutingTables\ \text{EXCEPT}\ ![i] = nrt] \\
\qquad\qquad\quad \land\, LeafSets' = [LeafSets\ \text{EXCEPT}\ ![i] = nls] \\
\qquad\qquad\quad \land\, Probing' = [Probing\ \text{EXCEPT}\ ![i] = prb] \\
\qquad\qquad\quad \land\, MessagePool' = (MessagePool \setminus \{m\}) \cup msg \\
\qquad\qquad\quad \land\, Responsible' = [Responsible\ \text{EXCEPT}\ ![i] = ls.node] \\
\qquad\qquad\quad \land\, \text{UNCHANGED}\ \langle Status,\ ToJoin,\ Failed \rangle
\end{array}
$$

Similarly, the action *ReceiveProbeReply* had no effect on the variables *Leases* and *Grants* in the original specification. However, it is modified so that on receiving the last expected probe reply (finishing the probing phase), a Waiting node i turns Ready, and, instead of sending lease request messages to its neighbors, notifies its responsible node that it has finished joining. The variable *Responsible* remains unchanged.

Definition 41 (Simplified LuPastry$^+$ Action Receive Probe Reply)**.**

$$
\begin{array}{l}
ReceiveProbeReply(i) \triangleq Status[i] \neq \text{"Dead"} \wedge \exists\, m \in MessagePool : \\
\quad \wedge\, m.content.type = \text{"ProbeReply"} \wedge m.destination = i \\
\quad \wedge\, \text{LET } j \triangleq m.content.node \\
\qquad\quad ls \triangleq m.content.ls \\
\qquad\quad nf \triangleq Failed[i] \setminus \{j\} \\
\qquad\quad nls1 \triangleq AddToLS(\{j\}, LeafSets[i]) \\
\qquad\quad nls2 \triangleq AddToLS((LeafSetContent(ls) \setminus nf), nls1) \\
\qquad\quad pm \triangleq LeafSetContent(nls2) \\
\qquad\quad prb1 \triangleq (LeafSetContent(nls1) \cap m.content.failed) \\
\qquad\qquad\qquad \setminus (Probing[i] \cup nf \cup \{i\}) \\
\qquad\quad prb2 \triangleq pm \setminus (LeafSetContent(nls1) \cup Probing[i] \cup nf \cup prb1) \\
\qquad\quad prb3 \triangleq (Probing[i] \cup prb1 \cup prb2) \setminus \{j\} \\
\qquad\quad fin \triangleq Status[i] = \text{"Waiting"} \wedge prb3 = \{\} \\
\qquad\quad msg \triangleq [destination \mapsto Responsible[i], \\
\qquad\qquad\qquad content \mapsto [type \mapsto \text{"Notify"}, node \mapsto i]] \\
\quad \text{IN} \quad \wedge\, RoutingTables' = [RoutingTables \text{ EXCEPT } ![i] = \\
\qquad\qquad\qquad\qquad AddToRT(\{j\}, @, i)] \\
\qquad\quad \wedge\, LeafSets' = [LeafSets \text{ EXCEPT } ![i] = nls1] \\
\qquad\quad \wedge\, Failed' = [Failed \text{ EXCEPT } ![i] = \text{IF } fin \text{ THEN } \{\} \text{ ELSE } nf] \\
\qquad\quad \wedge\, Probing' = [Probing \text{ EXCEPT } ![i] = prb3] \\
\qquad\quad \wedge\, Status' = [Status \text{ EXCEPT } ![i] = \\
\qquad\qquad\qquad\qquad \text{IF } fin \text{ THEN "Ready" ELSE } @] \\
\qquad\quad \wedge\, MessagePool' = (MessagePool \\
\qquad\qquad \cup\, ProbeSet(i, nls1, nf, prb1) \cup ProbeSet(i, nls1, nf, prb2) \\
\qquad\qquad \cup\, \text{IF } fin \text{ THEN } \{msg\} \text{ ELSE } \{\}) \setminus \{m\} \\
\qquad\quad \wedge\, \text{UNCHANGED } \langle Responsible, ToJoin \rangle
\end{array}
$$

Actions *RequestLease*, *ReceiveLeaseRequest* and *ReceiveLeaseReply* from the original specification are no longer needed and are omitted here. Instead, I define an action *ReceiveNotification*, where a node i receives a notification from its to-join node, letting it know that it has finished the join process and turned Ready. Node i resets its to-join field so that it can help new nodes join the network in the future.

Definition 42 (Simplified LuPastry$^+$ Action Receive Notification)**.**

$$
\begin{aligned}
&ReceiveNotification(i) \triangleq Status[i] = \text{"Ready"} \land \exists\, m \in MessagePool : \\
&\quad \land\, m.content.type = \text{"Notify"} \land m.destination = i \\
&\quad \land\, m.content.node = ToJoin[i] \land ToJoin' = [ToJoin \text{ EXCEPT } ![i] = i] \\
&\quad \land\, MessagePool' = (MessagePool \setminus \{m\}) \\
&\quad \land \text{ UNCHANGED } \langle Status, RoutingTables, LeafSets, Probing, Failed, \\
&\qquad\qquad\qquad Responsible \rangle
\end{aligned}
$$

This concludes the list of changes to the specification of LuPastry$^+$. In what follows, I describe the similar changes that had to be made to the proof to adapt it to the simplified specification.

7.3 Correct Delivery in Simplified LuPastry$^+$

Adapting the proof of correct delivery for the original LuPastry$^+$ specification to the simplified version is straight-forward. In fact, the proof described in Chapter 6 focuses mainly on the probing process and proves properties relating to the set of Ready/OK nodes as a whole, instead of just the set of Ready nodes.

In Chapter 6, I show that the set of Ready/OK nodes in a LuPastry$^+$ network is always stable, since every Waiting node must probe or be probed by its left and right Ready/OK neighbor before it turns OK. That is, the probing phase already establishes (and maintains) stability for both Ready and OK nodes.

Eliminating the lease exchange phase and the intermediate OK status in Simplified LuPastry$^+$ effectively reduces the set of Ready/OK nodes to the set of Ready nodes. Therefore, we can show that the set of Ready nodes are always stable in Simplified LuPastry$^+$, due to the probing phase.

Since exclusive coverage and correct delivery follow from stability, as shown in the reduction proof of Section 6.4, it is straight-forward to prove (strong) correct delivery for Simplified LuPastry$^+$. The property proven is the same as in Definition 2 in the previous chapter.

$$
\begin{aligned}
&\forall\, i, j, k \in I : \\
&\quad \land\, Status[i] = \text{"Ready"} \land Status[j] = \text{"Ready"} \\
&\qquad \land\, j \neq i \land Covers(LeafSets[i], k) \\
&\quad \land\, \exists\, m \in MessagePool : m.content.type = \text{"Lookup"} \\
&\qquad \land\, m.content.node = k \land m.destination = i \\
&\Rightarrow \land\, AbsoluteDistance(i, k) \leq AbsoluteDistance(j, k) \\
&\quad \land\, \neg Covers(LeafSets[j], k)
\end{aligned}
$$

Adapting the TLA$^+$ proof required the following steps.

1. *Removing invariants.* Invariants relating to the lease exchange phase were no longer needed and therefore eliminated. These are invariants relating to either exchanged lease request and reply messages, or the variables *Leases* and *Grants*, like Invariant 9, which says that for any node i, all nodes in the set $Leases[i]$ are Ready/OK. Since the proof of other invariants is mostly independent from invariants on lease exchange, this step did not cause any significant disturbance to the flow of the overall proof.

2. *Modifying invariants.* The proof of the remaining invariants had to be adjusted for the new next-state relation. Induction steps for the eliminated actions *RequestLease*, *ReceiveLeaseRequest* and *ReceiveLeaseReply* were removed. For the new action *ReceiveNotification*, one induction step is added.

3. *Adding invariants.* A few very simple invariants were added as necessary, to prove some properties about the new variable *Responsible*, such as Invariant 51 below.

Invariant 51. *Let i be a to-join node that is not Ready. If i's leaf set is not empty, then i's Responsible field is not set to itself.*

$$\forall\, i \in (ToJoinNodes \setminus ReadyNodes) :$$
$$\quad LeafSets[i] \neq EmptyLS(i) \Rightarrow i \neq Responsible[i]$$

In summary, it was possible to adapt the TLA$^+$ specification of LuPastry$^+$ described in Chapter 4 to have a simpler join process, and prove correct delivery for this join process by introducing only a small number of changes to the proof presented in Chapter 6.

Chapter 8

Conclusion

This thesis is concerned with the formal verification of Pastry, a popular implementation of a Distributed Hash Table (DHT). A DHT is a peer-to-peer network that acts as a hash table where different key-value pairs are stored at different nodes on the network. Pastry nodes are randomly assigned identifiers from the key space of the hash table. Each node is responsible for managing the portion of the key space numerically closest to its identifier. Like a classic hash table, the main function provided by Pastry is key lookup: a node may issue a lookup request for a certain key, and the request is routed to the Pastry node that manages this key. Pastry is a dynamic network where nodes may join the network at any time by following the join protocol, and leave/fail spontaneously without giving notice to other nodes.

The work presented here is a continuation of Tianxiang Lu's work on verifying the correct delivery of lookup requests in Pastry. Lu defines "correct delivery" as follows: "at any point in time, there is at most one node that answers a lookup request for a key, and this node must be the closest live node to that key." He shows that the full published version of Pastry violates correct delivery, and presents LuPastry, a pure-join variant of Pastry where node failure is not modeled and where a live node on the network may help at most one other node join the network at a time. Lu also gives a partial proof of correct delivery for LuPastry, mechanized in the TLA$^+$ proof assistant.

Upon examining Lu's proof, I found that it is based on a large number of unproven assumptions. This is likely due to the sheer size of the proof (over 10,000 lines) and the lack of maturity of the TLA$^+$ proof assistant at the time. In attempting to prove these assumptions, I found counterexamples to many of them, such as arithmetic assumptions or assumptions about protocol data structures that ignoring border cases. As a consequence, I was able to find a counterexample to one of Lu's claimed invariants of LuPastry which he uses to prove correct delivery. These discoveries highlighted the need for a new and complete proof of correct delivery in LuPastry.

The contribution of this thesis is manyfold.

1. I present LuPastry$^+$, a TLA$^+$ specification based on Lu's specification of

LuPastry, which introduces a number of bug fixes and improvements to the definitions of some operators. The improvements make the specification more readable and modular, and significantly improve the automation of the correctness proof.

2. I present a complete proof of correct delivery of lookup requests for LuPastry$^+$ in TLA$^+$.

3. I observe that the final step of the node join process of LuPastry/LuPastry$^+$, denoted by the "lease exchange" step, is in fact unnecessary for correct delivery. I present a simplified version of LuPastry$^+$ where the lease exchange step is eliminated. I denote the simplified specification by Simplified LuPastry$^+$.

4. I present a complete proof of correct delivery of lookup requests for Simplified LuPastry$^+$ in TLA$^+$. This proof is adapted from the TLA$^+$ proof for LuPastry$^+$, by introducing the necessary changes.

The lease exchange phase eliminated in Simplified LuPastry$^+$ was, in fact, not published as part of the original specification of Pastry, but in a later paper by the authors [21]. It was included by Lu in his specification of LuPastry in his attempt to devise a variant of Pastry for which correct delivery can be proven [27]. Lu shows that in the full specification of Pastry where node failure is modeled, the lease exchange phase does not prevent the protocol from violating the correctness property, since network separation is still possible if too many nodes fail within a small period of time. The proof presented here shows that, in fact, this phase of the join process is not necessary for correctness in the pure-join model.

Aside from the necessary technical differences, both proofs presented here have the same basic idea, and are comparable in size w.r.t. the number of interactive proof steps and the number of invariants proven. Each proof consists of over 32,000 lines. To my knowledge, both Lu's partial proof and the two proofs presented here represent the only two works that employ full theorem proving for formal verification of a DHT protocol, and the proofs presented here may well be the largest proof examples written using the TLA$^+$ proof assistant.

Experience and Challenges

Since both TLA$^+$ specifications of LuPastry$^+$ and Simplified LuPastry$^+$ are based on Lu's TLA$^+$ specification of LuPastry, I was spared the initial task of creating a specification from scratch, with the challenges that this task entails, such as determining the right level of abstraction, designing the data structures, and resolving ambiguities in the informal specifications found in the literature. In fact, my starting point was analyzing Lu's partial proof, and attempting to complete it by proving the unproven assumptions. It is through this task

that I learned the TLA$^+$ specification and proof language, also consulting the available references from time to time.

I found the TLA$^+$ language natural, expressive, and well-suited for modeling and verifying Pastry. The learning curve was not too long; within the first months I was sufficiently competent in TLA$^+$ to write involved proofs for some of Lu's assumptions, and, more importantly, to identify room for improvement in Lu's specification to enhance proof automation. Due to the expressiveness of the language, it is typically possible to express one property in many different ways that are semantically equivalent, but where the performance of the back-end theorem provers varies greatly according to the chosen syntax. For me, this was the first main task which gave me insight into both TLA$^+$ and Pastry. At this point, I decided to improve the specification of LuPastry into LuPastry$^+$ as described in Chapter 5, before proceeding to write a new and complete proof of correctness.

The second task was to prove a large number of lemmas on arithmetic, set theory and the Pastry data structures. Many of the data structure lemmas were inspired by Lu's unproven assumptions. The arithmetic and set theory lemmas, on the other hand, as well as many more data structure lemmas, were formulated and proven on-the-go whenever the need for them arose in the higher levels of the proof.

The most difficult task was to develop a coherent story for the proof and find the right invariants to prove. Model checking was crucial at this point. I performed model checking on networks of four and sometimes eight nodes for every property I intended to prove as an invariant. For the model checking to be efficient, the key trick is to model-check a restricted specification where actions for lookup and message loss are disabled.[1] Since the safety property "correct delivery" is concerned with the correctness of the node's local view of the network, these actions cannot contribute to any potential counterexamples. The lookup actions only move lookup messages through the network and do not change the node state. Similarly, message loss bears no effect on correct delivery; it can only affect liveness by causing a node to deadlock, waiting on a message that was lost in communication. With these actions disabled, the LuPastry$^+$ (and Simplified LuPastry$^+$) models have a finite number of states; the protocol stabilizes after every key in the key space corresponds to the identifier of a live node. At this point, all join processes have completed, and no more messages are exchanged. This means that model checking takes less than a minute on a network of four nodes instead of days. I found the TLC model checker easy to use from within the TLA$^+$ toolbox.

Another challenge in proving the invariants using TLAPS was the lack of counterexamples provided by the back-end provers. When proof obligations fail, the toolbox cites either timeout, or, rarely, a false proof obligation. In the case of a timeout, it takes some effort to determine whether the timeout occurred because the proof obligation was false, or because it was more than

[1] Actions *Lookup*, *RouteLookup*, *DeliverLookup* and *LoseMessage* in the TLA$^+$ specification presented in Chapter 4.

the back-end could handle. In my experience, many SMT-reported timeouts were due to a true proof obligation being too involved for the SMT solver to handle, while timeouts reported from general-purpose provers like Zenon and Spass typically meant the input proof obligation was false.

Room for Improvement and Future Work

Node failure. The proof presented here proves correct delivery of lookup messages for pure-join variants of Pastry, where node failure is not modeled. As Lu already shows in [27], the full published version of Pastry is inconsistent; the property of correct delivery is violated when node failure is taken into account. Similar results have been found for similar protocols like Chord [44]. In fact, it is argued that one cannot achieve consistency where node failure is allowed, due to the possibility of network separation [20]. However, there is still no definitive answer to this question. There is no doubt that a full DHT protocol that is formally specified and provably correct using rigorous formal methods would be a remarkable achievement in this research area. A good start would be Pamela Zave's version of Chord presented in [45], which she supports with a proof of correctness that is carried out by automated analysis of an Alloy model. Zave's model of Chord operates under the assumption that the key space has a certain minimum size n, and that there is a *stable base* of n nodes that are permanently online to guarantee connectivity. It would be interesting to apply formal methods like theorem proving to full specifications of Pastry or Chord which allow nodes to join and leave freely, and see if correct delivery holds under the assumption of a stable base.

Proof reuse for other DHT protocols. The strong similarities among the available DHT implementations indicate that there is great potential for proof reuse. This thesis already serves as an example in this direction, where a complete proof as large as 32,000 lines for LuPastry$^+$ was directly adapted to prove correctness of a simplified variant of the specification. There is also the possibility of applying formal methods to verify certain properties of a "generic" specification of DHT protocols, from which existing DHT protocols like Pastry can be instantiated. In his thesis, Ghodsi mentions that $\mathcal{DKS}$ is a generalization of DHT protocols, from which Pastry and Chord can be instantiated [20]. $\mathcal{DKS}$ may serve as a good starting point in this direction.

Bibliography

[1] Luc Onana Alima, Sameh El-Ansary, Per Brand, and Seif Haridi. A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid*, CCGrid 2003, pages 1–7, 2003.

[2] Noran Azmy. TLA$^+$ Specification and Proof Files of LuPastry, LuPastry$^+$ and Simplified LuPastry$^+$. `http://www.mpi-inf.mpg.de/departments/automation-of-logic/people/noran-azmy/`, 2015.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[4] Rana Bakhshi and Dilian Gurov. Verification of Peer-to-Peer Algorithms: A Case Study. *Electronic Notes in Theoretical Computer Science*, 181:35–47, June 2007.

[5] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV 2011, pages 171–177, 2011.

[6] Bernhard Beckert. Formal Specification and Verification. `http://formal.iti.kit.edu/beckert/teaching/Formale-Verifikation-SS09/01Intro.pdf`. Accessed: June 30, 2016.

[7] Fabian Benduhn, Thomas Thüm, Malte Lochau, Thomas Leich, and Gunter Saake. A Survey on Modeling Techniques for Formal Behavioral Verification of Software Product Lines. In *Proceedings of the 9th International Workshop on Variability Modeling of Software-intensive Systems*, VaMoS 2015, pages 80:80–80:87, New York, NY, USA, 2015. ACM.

[8] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR 2007, pages 151–165, 2007.

[9] Johannes Borgström, Uwe Nestmann, Luc Onana Alima, and Dilian Gurov. Verifying a Structured Peer-to-Peer Overlay Network: The Static Case. In *Global Computing, IST/FET International Workshop, Revised Selected Papers*, GC 2004, pages 250–265, 2004.

[10] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure Routing for Structured Peer-to-peer Overlay Networks. *SIGOPS Operating Systems Review*, 36:299–314, December 2002.

[11] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Exploiting Network Proximity in Peer-to-Peer Overlay Networks. Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

[12] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. One Ring to Rule Them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks. In *Proceedings of the 10th ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[13] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A Large-scale and Decentralized Application-level Multicast Infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499, September 2006.

[14] Microsoft Corporation. What Are P2P Communications? `https://support.skype.com/en/faq/FA10983/what-are-p2p-communications`. Accessed: June 19, 2016.

[15] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA$^+$ Proofs. In *18th International Symposium on Formal Methods*, volume 7436 of *LNCS*, pages 147–154, Paris, France, 2012. Springer.

[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS 2008/ETAPS 2008, pages 337–340, 2008.

[17] 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 the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP 2007, pages 205–220, 2007.

[18] Peter Druschel and Antony Rowstron. PAST: A Large-scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, HotOS VIII, pages 75–80, 2001.

[19] George H. L. Fletcher, Hardik A. Sheth, and Katy Börner. Unstructured Peer-to-Peer Networks: Topological Properties and Search Performance. In *Agents and Peer-to-Peer Computing: 3rd International Workshop, Revised and Invited Papers*, AP2PC 2004, pages 14–27, 2005.

[20] Ali Ghodsi. *Distributed K-ary System: Algorithms for Distributed Hash Tables*. PhD thesis, 2006.

[21] Andreas Haeberlen, Jeff Hoye, Alan Mislove, and Peter Druschel. Consistent Key Mapping in Structured Overlays. Technical Report TR05-456, Rice University, Department of Computer Science, August 2005.

[22] Adobe Systems Incorporated. What Are Peer-Assisted Networking Settings? `https://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager.html#117802`. Accessed: June 19, 2016.

[23] M. Eric Johnson, Dan McGuire, and Nicholas D. Willey. The Evolution of the Peer-to-Peer File Sharing Industry and the Security Risks for Users. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences*, page 383, 2008.

[24] Leslie Lamport. A Summary of TLA$^+$. `http://research.microsoft.com/en-us/um/people/lamport/tla/summary-standalone.pdf`. Accessed: July 27, 2016.

[25] Leslie Lamport. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, MA, USA, 2002.

[26] Leslie Lamport. The TLA$^+$ Hyperbook. `http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html`, 2015. Accessed: July 27, 2016.

[27] Tianxiang Lu. *Formal Verification of the Pastry Protocol*. PhD thesis, 2013.

[28] Tianxiang Lu. Formal Verification of the Pastry Protocol Using TLA$^+$. In *1st Symposium on Dependable Software Engineering: Theories, Tools and Applications*, volume 9409 of *LNCS*, pages 284–299, 2015.

[29] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Model Checking the Pastry Routing Protocol. In *Proceedings of the 10th International Workshop Automatic Verification of Critical Systems*, AVOCS 2010, 2010.

[30] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Towards Verification of the Pastry Protocol Using TLA$^+$. In *Formal Techniques for Distributed Systems*, pages 244–258, 2011.

[31] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlays. In *Peer-to-Peer Systems II: 2nd International Workshop, Revised Papers*, IPTPS 2003, pages 21–32. Springer, 2003.

[32] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *IPTPS '01 Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 53–65, 2002.

[33] Stephan Merz. The Specification Language TLA$^+$. In *Logics of Specification Languages*, pages 401–451. Springer, 2008.

[34] Palo Alto Networks. Application Usage & Threat Report. `http://researchcenter.paloaltonetworks.com/app-usage-risk-report-visualization`. Accessed: June 19, 2016.

[35] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-addressable Network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM 2001, pages 161–172. ACM, 2001.

[36] Anthony Rose. Evolution of the BBC iPlayer. `https://tech.ebu.ch/docs/techreview/trev_2008-Q4_iPlayer.pdf`, 2008. Accessed: July 12, 2016.

[37] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218 of *LNCS*, pages 329–350, November 2001.

[38] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM 2001, pages 149–160. ACM, 2001.

[39] TorrentFreak. Spotify Starts Shutting Down Its Massive P2P Network. `https://torrentfreak.com/spotify-starts-shutting-down-its-massive-p2p-network-140416/`, 2014. Accessed: July 12, 2016.

[40] Liang Wang and Jussi Kangasharju. Measuring Large-Scale Distributed Systems: Case of BitTorrent Mainline DHT. In *P2P*, pages 1–10. IEEE, 2013.

[41] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS Version 3.5. In *Proceedings*

of the 22nd International Conference on Automated Deduction, CADE-22, pages 140–145.

[42] Beverly Yang and Hector Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB 2001, pages 561–570. Morgan Kaufmann Publishers Inc., 2001.

[43] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In *Correct Hardware Design and Verification Methods (CHARME 1999)*, volume 1703 of *LNCS*, pages 54–66, 1999.

[44] Pamela Zave. Using Lightweight Modeling to Understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49, March 2012.

[45] Pamela Zave. How to Make Chord Correct (Using a Stable Base). Computing Research Repository (CoRR), 2015.

[46] Hao Zhang, Yonggang Wen, Haiyong Xie, and Nenghai Yu. A Survey on Distributed Hash Table (DHT): Theory, Platforms, and Applications. `http://www.ntu.edu.sg/home/ygwen/Paper/ZWYX-13.pdf`, 2013. Accessed: July 27, 2016.

[47] Yuqing Zhou. On the Performance of P2P Network: An Assortment Method. *ArXiv e-prints*, September 2011.