Formal verification of the Pastry protocol

Tianxiang Lu

▶ To cite this version:

Tianxiang Lu. Formal verification of the Pastry protocol. Other [cs.OH]. Université de Lorraine, 2013. English. NNT : 2013LORR0179 . tel-01750356

HAL Id: tel-01750356
https://hal.univ-lorraine.fr/tel-01750356v1

Submitted on 29 Mar 2018

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.

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

Formal Verification of the Pastry Protocol

Tianxiang Lu

Dissertation zur Erlangung des Grades
des Doktors der Naturwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Saarbrücken
2013

Tag des Kolloquiums	27. November 2013
Dekan	Univ.-Prof. Dr. Mark Groves
Vorsitzender des Prüfungsausschusses	Prof. Dr. Holger Hermanns
Berichterstatter	Prof. Dr. Dominique Méry
	Prof. Dr. Roland Meyer
	Prof. Dr. Achour Mostéfaoui
	Prof. Dr. Stephan Merz
	Prof. Dr. Philippe Quéinnec
	Prof. Dr. Peter H. Schmitt
	Prof. Dr. Christoph Weidenbach
Akademischer Mitarbeiter	Dr. Lei Song

UNIVERSITÉ DE LORRAINE

UNIVERSITÄT DES SAARLANDES

Département de formation doctorale en informatique
École doctorale IAEM Lorraine

Vérification formelle du protocole Pastry

THÉSE

présentée et soutenue publiquement le 27 novembre 2013
pour l'obtention du

Doctorat de l'Université Henri Poincaré – Nancy 1
(spécialité informatique)

par

Tianxiang Lu

Président du jury:
Dominique Méry (professeur, Univ. de Lorraine)

Rapporteurs:
Philippe Quéinnec (professeur, ENSEEIHT, Toulouse)
Peter H. Schmitt (professeur, KIT, Karlsruhe)

Examinateurs:
Dominique Méry (professeur, Univ. de Lorraine)
Roland Meyer (professeur, TU Kaiserslautern)
Achour Mostéfaoui (professeur, Univ. de Nantes)

Directeurs:
Stephan Merz (directeur de recherche, Inria Nancy)
Christoph Weidenbach (research leader, Max-Planck-Institut für Informatik, Saarbrücken)

Laboratoire Lorrain de Recherche en Informatique et ses Applications – UMR 7503
Max-Planck-Institut für Informatik

Loria Laboratoire lorrain de recherche en informatique et ses applications

Inria INVENTEURS DU MONDE NUMÉRIQUE

mpii

Abstract

Pastry is a structured *P2P* algorithm realizing a Distributed Hash Table (*DHT*) over an underlying virtual ring of nodes. Hash keys are assigned to the numerically closest node, according to their Ids that both keys and nodes share from the same Id space. Nodes join and leave the ring dynamically and it is desired that a lookup request from arbitrary node for a key is routed to the responsible node for that key which then delivers the message as answer.

Several implementations of Pastry are available and have been applied in practice, but no attempt has so far been made to formally describe the algorithm or to verify its properties. Since Pastry combines rather complex data structures, asynchronous communication, concurrency, resilience to *churn*, i.e. spontaneous join and departure of nodes, it makes an interesting target for verification.

This thesis formally models and improves Pastry's core algorithms, such that they provide the correct lookup service in the presence of churn and maintain a local data structures to adapt the dynamic updates of neighborhood.

This thesis focuses on Join protocol of Pastry and formally defines different statuses (from "dead" to "ready") of a node according to its stage during join. Only "ready" nodes are suppose to have consistent key mapping among each other and are allowed to deliver the answer message. The correctness property is identified by this thesis to be *CorrectDelivery*, stating that there is always at most one node that can deliver an answer to a lookup request for a key and this node is the numerically closest "ready" node to that key. This property is non-trivial to preserve in the presence of *churn*.

The specification language TLA$^+$ is used to model different versions of Pastry algorithm starting with CastroPastry, followed by HaeberlenPastry, IdealPastry and finally LuPastry. The TLA$^+$ model checker TLC is employed to validate the models and to search for bugs. Validation ensures that the system has at least some useful executions; model analysis helps to discover unexpected corner cases to improve the model. Models are simplified for more efficient checking with TLC and consequently mitigating the state explosion problem.

Through this thesis, unexpected violations of *CorrectDelivery* in CastroPastry and HaeberlenPastry are discovered and analyzed. Based on the analysis, Haeberlen-Pastry is improved to a new design of the Pastry protocol IdealPastry, which is first verified using the interactive theorem prover TLAPS for TLA$^+$. IdealPastry assumes that a "ready" node handles one joining node at a time and it assumes that (1) no departure of nodes (2) no concurrent join between two "ready" nodes closed to each other. The last assumption of IdealPastry is removed by its improved version LuPastry. In LuPastry, a "ready" node adds the joining node directly when it receives the join request and does not accepts any further join request until it gets the confirmation from

the current joining node that it is "ready". LuPastry is proved to be correct w.r.t. *CorrectDelivery* under the assumption that no nodes leave the network, which cannot be further relaxed due to possible network separation when particular nodes simultaneously leave the network.

The most subtle part of the deductive system verification is the search for an appropriate inductive invariant which implies the required safety property and is inductively preserved by all possible actions. The search is guided by the construction of the proof, where TLC is used to discover unexpected violations of a hypothetical invariant postulated in an earlier stage. The final proof of LuPastry consists of more than 10,000 proof steps, which are interactively checked in time by using TLAPS launching different back-end automated theorem provers.

This thesis serves also as a case study giving the evidence of possibility and the methodology of how to formally model, to analyze and to manually conduct a formal proof of complex transition system for its safety property. Using LuPastry as template, a more general framework on verification of *DHT* can be constructed.

Zusammenfassung

Pastry ist ein P2P (*peer-to-peer*) Algorithmus, der eine verteilte Hashtabelle (DHT) über einem als virtuellen Ring strukturierten Netzwerk realisiert. Knoten-Identifikatoren und Hash-Schlüssel entstammen derselben Menge, und jeder Knoten verwaltet die Schlüssel, die seinem Identifikator am nächsten liegen. Knoten können sich zur Laufzeit ins Netz einfügen bzw. es verlassen. Dennoch sollen Anfragen nach einem Schlüssel von beliebigen Knoten immer zu demjenigen Knoten weitergeleitet werden, der den Schlüssel verwaltet und der die Anfrage dann beantwortet.

Pastry wurde mehrfach implementiert und praktisch eingesetzt, aber der Algorithmus wurde bisher noch nie mathematisch präzise modelliert und auf Korrektheit untersucht. Da bei Pastry komplexe Datenstrukturen, asynchrone Kommunikation in einem verteilten Netzwerk und Robustheit gegen *churn*, d.h. spontanes Einfügen oder Verlassen von Knoten zusammenkommen, stellt das Protokoll eine interessante Fallstudie für formale Verifikationstechniken dar.

In dieser Arbeit werden die Kernalgorithmen von Pastry modelliert, die Anfragen nach Schlüsseln in Gegenwart von *churn* behandeln und lokale Datenstrukturen verwalten, welche die jeweiligen Nachbarschaftsbeziehungen zwischen Knoten zur Laufzeit widerspiegeln.

Diese Dissertation behandelt insbesondere das Join-Protokoll von Pastry zum Einfügen neuer Knoten ins Netz, das jedem Knoten seinen Status (von "dead" bis "ready") zuweist. Knoten mit Status "ready" müssen untereinander konsistente Modelle der Zuständigkeit für Schlüssel aufweisen und dürfen Anfragen nach Schlüsseln beantworten. Als zentrale Korrektheitseigenschaft wird in dieser Arbeit *CorrectDelivery* untersucht, die ausdrückt, dass zu jeder Zeit höchstens ein Knoten Anfragen nach einem Schlüssel beantworten darf, und dass es sich dabei um den Knoten mit Status "ready" handelt, dessen Identifikator dem Schlüssel numerisch am nächsten liegt. In Gegenwart von *churn* ist es nicht einfach diese Eigenschaft sicherzustellen.

Wir benutzen die Spezifikationssprache TLA$^+$, um verschiedene Versionen des Pastry-Protokolls zu modellieren: zunächst CastroPastry, gefolgt von HaeberlenPastry und IdealPastry, und schließlich LuPastry. Mit Hilfe des Modelcheckers TLC für TLA$^+$ werden verschiedene qualitative Eigenschaften untersucht, um die Modelle zu validieren und Fehler zu finden. Dafür werden die Modelle zum Teil vereinfacht, um das Problem der Zustandsexplosion zu mindern und so die Effizienz des Modelchecking zu verbessern.

Diese Arbeit konnte unerwartete Abläufe von CastroPastry und HaeberlenPastry aufdecken, bei denen die Eigenschaft *CorrectDelivery* verletzt wird. Auf der Grundlage dieser Analyse und einiger Verbesserungen von HaeberlenPastry wird das Protokoll IdealPastry entwickelt und seine Korrektheit mit Hilfe des interaktiven

Theorembeweisers TLAPS für TLA$^+$ gezeigt. Das Protokoll IdealPastry stellt sicher, dass ein "ready" Knoten zu jeder Zeit höchstens einen neuen Knoten ins Netz einfügt, und es nimmt an, dass (1) kein Knoten je das Netz verlässt und (2) keine zwei Knoten zwischen benachbarten "ready" Knoten eingefügt werden. Der Algorithmus LuPastry verbessert IdealPastry und beseitigt Annahme (2) von IdealPastry. In dieser Version nimmt ein "ready" Knoten den neu einzufügenden Knoten unmittelbar in seine Nachbarschaft auf und akzeptiert dann solange keinen weiteren neu hinzukommenden Knoten, bis der erste Knoten bestätigt, dass er Status "ready" erreicht hat. LuPastry wird als korrekt bezüglich der Eigenschaft *CorrectDelivery* nachgewiesen, unter der Annahme, dass keine Knoten das Netz verlassen. Diese Annahme kann im allgemeinen nicht vermieden werden, da der Ring in separate Teilnetze zerfallen könnte, wenn bestimmte Knoten gleichzeitig das Netz verlassen.

Die größte Herausforderung bei deduktiven Ansätzen zur Systemverifikation ist es, eine geeignete Invariante zu finden, die sowohl die angestrebte Sicherheitseigenschaft impliziert als auch induktiv von allen Systemaktionen erhalten wird. Während der Konstruktion des Korrektheitsbeweises wird TLC benutzt, um unerwartete Gegenbeispiele zu hypothetischen Invarianten zu finden, die zuvor postuliert wurden. Der Beweis des LuPastry-Protokolls besteht aus mehr als 10000 Beweisschritten, die von TLAPS und seinen integrierten automatischen Theorembeweisern verifiziert werden.

Die vorliegende Arbeit dient auch als Fallstudie, welche die Möglichkeit der formalen Modellierung, Analyse und Korrektheitsbeweises von komplexen Transitionssystemen aufzeigt und die dabei notwendigen Einzelschritte und -techniken behandelt. LuPastry kann als Vorlage benutzt werden, um einen allgemeineren Rahmen für die Verifikation von DHT-Protokollen zu schaffen.

Résumé

Pastry est un algorithme qui réalise une table de hachage distribuée (THD) sur un réseau pair à pair organisé en un anneau virtuel de nœuds. Chaque nœud gère les informations dont les clés sont numériquement proches de son propre identifiant, sachant que les espaces d'identifiants de nœuds et de clés sont confondus dans ce protocole. Le protocole admet que des nœuds puissent rejoindre ou quitter l'anneau à tout moment. Il doit néanmoins faire suivre toute requête pour une clé, provenant de n'importe quel nœud, au nœud qui est responsable pour cette clé et qui répondra par l'information recherchée.

Il existe plusieurs implémentations de Pastry qui ont été utilisées en pratique. Cependant, il n'y a pas encore eu de travaux qui visent à décrire formellement l'algorithme ou à vérifier son bon fonctionnement. Intégrant des structures de données complexes, de la communication asynchrone dans un contexte d'un protocole réparti et une robustesse vis-à-vis du *churn*, i.e. des nœuds qui rejoignent ou quittent le réseau, ce protocole représente un intérêt certain pour être analysé par des techniques de vérification formelle.

Dans cette thèse nous modélisons les algorithmes au cœur de Pastry et qui réalisent le service de recherche d'un nœud responsable d'une clé en présence de *churn*. Ces algorithmes maintiennent en particulier une structure locale de données pour gérer les évolutions dynamiques de la relation de voisinage.

La thèse étudie en particulier le protocole Join de Pastry qui permet à un nœud de rejoindre le réseau et qui associe un statut (variant entre ≪mort≫et ≪prêt≫) à tout nœud. Les nœuds ≪prêts≫doivent avoir une vue du voisinage cohérente entre eux ; ils sont autorisés à répondre à des messages de recherche d'informations. La propriété principale de correction qui nous intéresse ici, appelée *CorrectDelivery*, assure qu'à tout moment il y a au plus un nœud capable de répondre à une requête pour une clé, et que ce nœud est le nœud le plus proche numériquement à ladite clé. Il n'est pas trivial de maintenir cette propriété en présence de *churn*.

Le langage de spécification formelle TLA$^+$ est utilisé pour modéliser différentes versions du protocole Pastry, en commençant par CastroPastry, suivant par HaeberlenPastry, IdealPastry, puis LuPastry. Le *model checker* TLC associé à TLA$^+$ sert pour valider des modèles et pour trouver des erreurs, en vérifiant des propriétés qualitatives. Pour ce faire, nous utilisons parfois des modèles simplifiés pour pallier au problème de l'explosion combinatoire du nombre d'états.

Ce travail nous a permis de découvrir des violations inattendues de la propriété *CorrectDelivery* dans CastroPastry et HaeberlenPastry. Sur la base de cette analyse et des améliorations apportées à HaeberlenPastry, le protocole IdealPastry et conçu et vérifié en utilisant l'assistant interactif à la preuve TLAPS pour TLA$^+$. IdealPastry est conçu de façon à ce que tout nœud ≪prêt≫ne gère qu'un nœud désirant

rejoindre le réseau à la fois, et il suppose que (1) aucun nœud ne quitte le réseau et (2) il n'y a jamais deux nœuds qui rejoignent le réseau en même temps entre deux nœuds «prêts»proches l'un de l'autre. Cette dernière hypothèse du protocole IdealPastry est levée dans sa version améliorée LuPastry. Dans cette version, un nœud «prêt»ajoute le nœud désirant rejoindre le réseau immédiatement dans son voisinage et n'accepte d'autre requête à rejoindre le réseau que si le premier nœud à confirmé qu'il est «prêt». Il est montré formellement que LuPastry vérifie *CorrectDelivery* sous l'hypothèse qu'aucun nœud ne quitte le réseau. Cette hypothèse ne peut être relâchée à cause du risque de perte de connexion du réseau dans le cas où plusieurs nœuds spécifiques quittent le réseau en même temps.

La tâche la plus ardue en vérification déductive est de trouver un invariant inductif adéquat qui à la fois implique la propriété de sûreté que l'on souhaite démontrer et est préservé par toute action du protocole. Cette tâche est guidée par la construction de la preuve formelle, et l'utilisation de TLC permet de découvrir des violations inattendues d'invariants hypothétiques postulés au préalable. La preuve finale de LuPastry consiste en environ 10000 étapes de preuve qui sont vérifiées par TLAPS et ses différents outils automatiques de preuve.

Par le biais d'une étude de cas conséquente, cette thèse met en évidence la possibilité et étudie la méthodologie pour la modélisation d'un système de transitions complexe et pour son analyse et vérification déductive formelle en vue d'établir une propriété de sûreté. En se servant de LuPastry comme une calque, un cadre plus général de vérification de THD peut être envisagé.

摘要

Pastry 是一个将分布式哈希表（DHT）实现在虚拟环结构上的点对点协议的结构化算法。在这类算法中，哈希密匙（Hash Key）和网络节点共享一个ID域，而密匙被则被分配在和它ID距离最近的网络节点上。网络节点可能随时加入或离开，而该协议正确的预期效果之一就是能够为任意的寻址操作提供正确的路由以便得到正确的答案。

Pastry 有很多版本并且已经被广泛应用了，但迄今为止，对其算法及正确寻址等性质的形式化描述或验证尚未有任何进展。Pastry 综合了复杂的数据结构、异步通讯、并发性、抗震荡性（震荡, *churn*, 这里指由节点的动态加入和离开所带来的网络拓扑结构的改变）等多种性质，因而对它的形式化验证成为了很有科学价值的研究课题。

本博士论文对Pastry的核心算法进行了建模，并改善了协议的设计，最终证明了改善的模型在网络震荡的环境下能够正确寻址。

论文重点分析了算法中节点的加入协议并定义了节点在加入的流程中从断连（dead）到准备就绪（ready）的不同状态。仅有准备就绪的节点才能保证一致的哈希映射，并且只有它们能答复寻找密匙的请求。协议的正确性被形式化的描述为*CorrectDelivery*, 即在网络运行的任意时刻，最多有一个节点能够向寻址操作提供结果，且该点应当是所有完备节点中离密匙最近的节点。在网络震荡频发的状态下，这个性质是很难维持的。

TLA$^+$是一种规约语言，在本文中用于对不同版本的Pastry算法进行建模，其中包括初始的 CastroPastry, 到后继的 HaeberlenPastry，和 IdealPastry以及LuPastry。 TLA$^+$的模型检测工具TLC被用来确认模型符合设计初衷。为了更高效的使用该工具，模型被进一步简化，这样以来就避免了模型检测中常见的状态爆炸的问题。

本论文描述了在形式化分析 CastroPastry和 HaeberlenPastry时发现的的潜在问题，并基于对它们的改进，设计了第一个被形式化验证的Pastry协议 IdealPastry。该协议的设计建立在两个理想状态的假设上：（1）没有节点离开（2）没有并发节点在两个相邻准备就绪的节点间加入网络。第二条假设在 LuPastry中被去除了。LuPastry协议中的节点一旦收到加入请求就马上将相应节点加入本节点的邻居信息里，并不再处理之后的所有加入请求，直到收到该加入节点准备就绪的通知。LuPastry在没有节点离开的前提下最终被形式化验证为符合Pastry所期待的协议，该协议被验证为能够完全抵抗并发节点加入网络带来的震荡。

课题论证最大的挑战就是寻找并证明那些不变量，即网络在所有节点执行任意操作时保持其正确性的那些可以全局归纳的性质。寻找过程实际上是和证明过程同步进行的，这里 TLC用来对寻找过程中发现的备选不变量进行预期检测。最终的证明包含了一万多步骤，每一步骤都通过 TLAPS系统后台的自动证明机进行了验证。

本课题也可被看做是一次对使用形式化建模、分析并论证一个复杂网络协议之正确性的可行性及方法论的探索实例，这标示着目前的形式化验证技术已经可以应用在真实世界的分布式复杂算法中了。LuPastry可以用作模板，构建更通用的框架，以对其他分布式哈希表算法（DHT）进行验证。

Acknowledgements

The research that has gone into this thesis has been thoroughly enjoyable. This enjoyment is largely a result of the interaction that I have had with my colleagues at the Automation of Logics group of Max-Planck-Institute, the MOSEL group in LORIA and VeriDis group in INRIA, as well as the support by my mother and friends, who gave me invaluable mental support when I was suffering from a bad mood stemming from other things in my life.

First and foremost, I thank my two advisors Christoph Weidenbach and Stephan Merz, who accepted me as their *co-tutelle* doctoral student, for their constant support and for the time and energy they invested in our joint work. The guidance, encouragement and precious advice on both the research topic and the mentality of thinking and working rigorously have silently extended my character from only efficiency and goal oriented to placing a huge value on quality and methodology.

I thank Arnaud Fietzke, Willem Hagemann, Marek Kosta and Patrick Wischnewski for the countless informal discussions about mathematics, logics, philosophy and the lovely but often also annoying “rings of Pastry”. I have always considered such discussions as a major source of the joy of scientific work.

I thank the anonymous reviewers of the publications that are incorporated in this thesis. Their comments have helped me considerably in obtaining a fresh view on some problems and in improving both my results and my presentation thereof.

Of even greater help was the feedback from my colleague Noran Azmy, my friend Mathieu Flinders, my English teacher Alexei Kirk and my friend Cheng Li who proofread this thesis in different parts and various stages of its formation.

I also thank Leslie Lamport and Peter Druschel, who gave me encouragement and some guidance on my research as the inventors of TLA$^+$ and Pastry respectively.

I thank Dominique Méry, Roland Meyer, Achour Mostéfaoui, Philippe Quéinnec and Peter H. Schmitt for joining my thesis committee, as well as for interesting discussions and helpful comments.

This work has been supported by the International Max Planck Research School and by the German French University (DFH). I am thankful for their financial support.

Finally, I thank the (yet to come) readers of this thesis for their interest in my work. I sincerely hope that you can benefit from it.

Contents

0 Résumé étendu **1**
0.1 Motivation 1
0.2 Méthodologie 4
0.3 Notre preuve de Pastry 7
0.3.1 Défis dans la modélisation de CastroPastry et de HaeberlenPastry 8
0.3.2 Analyse des versions CastroPastry et HaeberlenPastry par model checking 8
0.3.3 Preuve de réduction 9
0.3.4 Conception et modélisation de IdealPastry 9
0.3.5 Validation de IdealPastry 10
0.3.6 Vérification de IdealPastry 10
0.3.7 Relâcher les hypothèses 11
0.3.8 Conception et modélisation de LuPastry 12
0.3.9 Validation de LuPastry 12
0.3.10 Vérification de LuPastry 12
0.4 Contributions de la thèse 12
0.5 Structure du mémoire de thèse 14

1 Introduction **15**
1.1 Motivation 15
1.2 Methodology 18
1.3 The Story of Verifying Pastry 20
1.3.1 Challenges of Modeling CastroPastry and HaeberlenPastry 21
1.3.2 Model Checking CastroPastry and HaeberlenPastry 22
1.3.3 Reduction Proof 22
1.3.4 Design and Modeling IdealPastry 22
1.3.5 Validation of IdealPastry 23
1.3.6 Verification of IdealPastry 23
1.3.7 Relaxing the Assumptions 24
1.3.8 Design and Modeling of LuPastry 25
1.3.9 Validation of LuPastry 25
1.3.10 Verification of LuPastry 25
1.4 Contributions of the Thesis 25
1.5 Structure of the Thesis 26

2 Foundations 29
2.1 Peer-to-peer System . . . 29
2.1.1 History, Concept and Definition . . . 30
2.1.2 Classification . . . 31
2.1.3 Structured Decentralized System and DHT . . . 32
2.2 Formal Verification . . . 34
2.2.1 Specification, Validation and Verification . . . 34
2.2.2 Model Checking and Theorem Proving . . . 36
2.3 TLA$^+$ Specification Language . . . 42
2.3.1 Constant Operators . . . 42
2.3.2 Modules . . . 44
2.3.3 Other TLA$^+$ Syntax . . . 46
2.3.4 Transition System . . . 46
2.3.5 Property, Inductive Invariant and Proof . . . 49
2.3.6 Basics of the TLA$^+$ Proof Language . . . 50

3 Pastry Protocol 53
3.1 Algorithms of Pastry . . . 53
3.1.1 Basic Idea of Pastry . . . 53
3.1.2 Notation . . . 57
3.1.3 Pseudocode of CastroPastry . . . 58
3.2 The Join Protocol of Pastry . . . 64
3.2.1 Concurrent Join Problem and CastroPastry . . . 65
3.2.2 Lease-Granting Protocol and HaeberlenPastry . . . 67
3.2.3 Related Work on Pastry . . . 69
3.2.4 Enhancements of the Protocol in IdealPastry and LuPastry . . 70
3.3 The Verified Pastry Join Protocol LuPastry . . . 75
3.3.1 Introduction with Flow Chart and Running Example . . . 76
3.3.2 Statuses of a Node . . . 80
3.3.3 Pseudocode of LuPastry . . . 81
3.3.4 Verification of LuPastry . . . 89
3.4 Assumptions . . . 91
3.5 Summary . . . 92

4 Detailed Formal Model of Pastry 93
4.1 Static Model . . . 93
4.1.1 Ring Operation . . . 93
4.1.2 Leafset . . . 94
4.1.3 Routing Table . . . 96
4.1.4 Messages . . . 98
4.2 Dynamic Model . . . 99
4.2.1 Lookup and Join . . . 101
4.2.2 Routing Actions . . . 102
4.2.3 Actions for Join Protocol . . . 105

4.2.4 Request and Grant Leases . . . 108
4.2.5 Statuses . . . 111
4.3 The Correctness Properties . . . 111
4.3.1 Type Correctness . . . 111
4.3.2 Safety Property . . . 112

5 Model Checking with TLC 115
5.1 Introduction . . . 115
5.2 Model Analysis on Pastry Properties . . . 117
5.2.1 Analysis of *CorrectDelivery* on CastroPastry . . . 117
5.2.2 Analysis of *NeighborProp* on HaeberlenPastry . . . 120
5.2.3 Analysis of Invariants of Final Model of Pastry . . . 123
5.3 Validation . . . 124
5.3.1 Successful Join and Deliver . . . 125
5.3.2 Successful Concurrent Join of Nodes . . . 125
5.3.3 Summary of Model Checking Results . . . 127
5.4 Experiences and Best Practices . . . 128

6 Formal Proof of the Property CorrectDelivery 131
6.1 Reducing *CorrectDelivery* to *NeighborProp* . . . 132
6.2 Towards an Inductive Proof of *NeighborProp* . . . 135
6.2.1 Inductive Proof of Invariant *HalfNeighbor* . . . 136
6.2.2 Towards an Inductive Proof of Invariant *NeighborClosest* . . . 139
6.3 A First Inductive Proof of *NeighborClosest* with Strong Assumptions . . . 143
6.3.1 Implementation of the Assumptions . . . 143
6.3.2 Invariants . . . 144
6.3.3 Proof Sketch of *CompleteLeafSet* as an Example . . . 148
6.4 Relaxing the Assumptions . . . 150
6.4.1 The Problem of Allowing Departure of Nodes . . . 151
6.4.2 The Approach of Allowing Concurrent Join . . . 152
6.4.3 The Meaning of Allowing Message Loss . . . 154
6.4.4 Modification of the Proof and Invariants . . . 154
6.5 Final Invariants and the Proof . . . 155
6.5.1 Invariants . . . 155
6.5.2 Proof Sketch of the Invariant *IRN* as an Example . . . 166
6.6 The TLA$^+$ Proof . . . 168

7 Related Works 175
7.1 Other *P2P* Systems . . . 175
7.1.1 Unstructured Decentralized Systems . . . 176
7.1.2 Hybrid (Partly Centralized) *P2P* Systems . . . 177
7.1.3 Summary of *P2P* Systems . . . 178
7.2 Formal Analysis of Chord . . . 178
7.2.1 Formal Analysis on Chord Using Alloy . . . 179

7.2.2 Previous Formal Approaches Analyzing Chord 180
7.3 Other Formal Methods Applied on Verifying Correctness of Network Protocol 181

8 Conclusion and Future Work 185
8.1 Answers to the Questions About Pastry 185
8.2 Answers to Questions About Methodology 187
8.3 Future Work 189

Bibliography 191

List of Figures

0.1 L'anneau virtuel de Pastry. . . . 3
0.2 Méthodologie de vérification en TLA$^+$. . . . 6

1.1 Pastry ring. . . . 17
1.2 Verification approach using TLA$^+$. . . . 20

2.1 Ontology of networks. . . . 31
2.2 Framework of formal verification. . . . 35
2.3 The model checking process. . . . 37
2.4 The interactive theorem proving process. . . . 41
2.5 Proof of Cantor's Theorem in TLA$^+$. . . . 51

3.1 Pastry routing example. . . . 55
3.2 Overview of the join protocol of CastroPastry. . . . 56
3.3 The ring configuration for the counterexample. . . . 66
3.4 Counterexample leading to a violation of *CorrectDelivery*. . . . 66
3.5 Extending the join protocol by lease granting. . . . 67
3.6 Node departure handling. . . . 69
3.7 The ignored "ok" node. . . . 72
3.8 Rejoin counterexample. . . . 73
3.9 Separation of the network due to simultaneous departures of nodes. . . . 74
3.10 Concurrent join with 5 nodes. . . . 75
3.11 Flow chart of complete join process of LuPastry. . . . 77
3.12 Join example: upgrades of the status. . . . 78

5.1 Screen shot of the TLA$^+$ Toolbox running model checker TLC. . . . 115
5.2 Violation trace of concurrent join in CastroPastry. . . . 119
5.3 Violation trace of departure and rejoin of nodes in HaeberlenPastry (part 1). . . . 121
5.4 Violation trace of departure and rejoin of nodes in HaeberlenPastry (part 2). . . . 122
5.5 Violation trace of property *Symmetry* in LuPastry. . . . 124

6.1 Case analysis for the proof sketch of *CoverageLemma* (Lemma 6.1.1). . . . 134
6.2 Case analysis for the proof sketch of *DisjointCovers* (Lemma 6.1.2). . . . 135
6.3 Intuition of hypothetical invariants *IncludeNeighbor* and *CrossNeighbor*. . 141
6.4 Counterexample of *IncludeNeighbor*. . . . 142

6.5 Hypothetical violation of *CompleteLeafSet* by its inductive proof. 149
6.6 Separation of the network due to concurrent departures of nodes. 151
6.7 Concurrent join with 5 nodes. 153
6.8 Hypothetical violation of *IRN* by its inductive proof. 172
6.9 TLA$^+$ codes of the inductive proof of invariants. 173

7.1 Different topologies of distributed systems. 175

0 Résumé étendu

Pastry (Rowstron and Druschel (2001), Castro et al. (2004), Haeberlen et al. (2005)) est un algorithme pair à pair (*P2P*) qui réalise une table de hachage distribuée (*DHT* Hellerstein (2003)) sur le support d'un réseau de recouvrement structuré en un anneau virtuel de nœuds. Plusieurs implémentations de Pastry existent et ont été utilisées en pratique mais à notre connaissance aucune tentative de description formelle de l'algorithme en vue d'une vérification de ses propriétés n'a jamais été entreprise. Puisque Pastry est une réalisation typique d'une *DHT* intégrant des structures de données assez complexes, de la communication asynchrone, du parallélisme et qui est supposée résister au *churn*, c'est à dire à l'arrivée et au départ concurrents de nœuds, cet algorithme présente un intérêt certain pour la vérification formelle.

Cette thèse modélise les algorithmes au cœur de Pastry qui fournissent un service de recherche de nœuds en présence de *churn* et maintiennent une structure de données locale pour tenir à jour l'information sur le voisinage de chaque nœud. Dans ce qui suit nous commençons par motiver nos intérêts de recherche, puis nous concrétisons les objectifs de ce travail, et nous expliquons comment ces objectifs sont atteints. Nous terminons par un aperçu de ce mémoire qui servira de guide pour le lire.

0.1 Motivation

La disruption d'un service logiciel tel que SKYPE (Microsoft (2013)) peut perturber notre vie quotidienne lorsqu'elle rend impossibles les appels des utilisateurs ou interrompt brusquement ces appels au milieu d'une conversation. SKYPE est très connu et utilisé mondialement par des milliards d'utilisateurs pour passer des appels téléphoniques à travers Internet. Le jeudi 16 août 2007, le réseau *P2P* à la base de SKYPE est devenu instable et a souffert d'une perturbation critique malgré la capacité supposée innée d'un réseau *P2P* à tolérer de telles perturbations. Arak (2007) affirme que ≪cet événement a révélé une erreur logicielle concernant l'algorithme d'allocation de ressources du réseau qui n'avait jamais été rencontrée auparavant et qui a empêché l'auto-réparation du réseau à fonctionner efficacement. ≫Malheureusement le 23 décembre 2010, des millions d'utilisateurs ont à nouveau été incapables de passer des appels, à nouveau à cause de départs de plusieurs ≪super-nœuds ≫. Existe-t-il de problèmes fondamentaux dans la conception de tels réseaux qui mènent invariablement à des arrêts intempestifs de temps à autre? Sinon, y a-t-il une preuve cohérente de la correction du protocole?

Il n'est pas connu publiquement comment SKYPE met en œuvre les idées d'algorithmes *P2P* pour la réalisation de ses services. Cependant, il ne fait pas de doute que la scalabilité de SKYPE est fondamentalement due à son adoption de protocoles *P2P*, lui

permettant de gérer des milliards d'appels téléphoniques à l'échelle mondiale, selon Microsoft (2013). Ainsi, les systèmes *P2P* ont gagné en popularité depuis le début du $21^{\text{ème}}$ siècle grâce à leurs propriétés d'auto-organisation et de décentralisation. Notamment, des systèmes *P2P* structurés réalisent des tables de hachage distribuées (*DHT*) qui sont censées fournir

- un routage efficace et fiable;
- une maintenance de la structure distribuée et à faible coût;
- et une robustesse à des arrivées et départs simultanés de nœuds du réseau.

C'est pourquoi les *DHT* servent souvent de base pour des systèmes répartis de grande échelle comme Dynamo (DeCandia et al. (2007)), une plate-forme de stockage commune à différents services au sein d'Amazon. Il est probable que Skype utilise une *DHT* pour que les pairs puissent trouver l'un l'autre correctement et efficacement.

La correction d'une *DHT* repose en grande partie sur l'algorithme qui la réalise. Le protocole Pastry est l'une des réalisations de *DHT* ayant trouvé une large utilisation.[1] Il réalise une *DHT* en affectant des clefs d'objets (par exemple des identifiants de données) à des nœuds du réseau de recouvrement (par exemple des ordinateurs connectés à Internet) et fournit la primitive *lookup* pour transmettre une requête au nœud qui gère la clef correspondante.

Questions autour de Pastry

Puisque Pastry réalise une *DHT* il est intéressant et crucial de comprendre les mécanismes fondamentaux de Pastry et d'analyser et de démontrer ses propriétés de correction. L'article Castro et al. (2004) introduit Pastry par du pseudo-code au niveau des messages échangés. En partant de cet article nous allons étudier les questions suivantes autour de Pastry:

- Comment fonctionne Pastry? En particulier, comment ce protocole réalise-t-il une *DHT*?
- Que veut dire précisément d'assurer un ≪routage fiable ≫? Est-ce que Pastry garantit la fiabilité du routage? Comment et à quel degré? Existe-t-il une erreur fondamentale de conception dans Pastry, et particulièrement dans la version présentée dans Castro et al. (2004), quant aux propriétés de correction, telles que le routage fiable? Si oui, comment ce problème peut-il survenir? Si non, y a-t-il une preuve formelle de la correction du protocole?
- Existe-t-il d'autres propriétés d'intérêt d'une telle réalisation d'une *DHT*? Ces propriétés sont-elles reliées les unes aux autres?

[1] Dans cette thèse nous distinguons différents niveaux d'abstraction. Nous employons le mot ≪réalisation ≫pour désigner le raffinement du concept de *DHT* à un algorithme concret comme Pastry. Le mot ≪implémentation ≫désigne différentes versions d'un algorithme. Les deux mots font référence à différents niveaux de détail dans la conception, et nous n'utilisons aucun de ces mots pour désigner un logiciel exécutable.

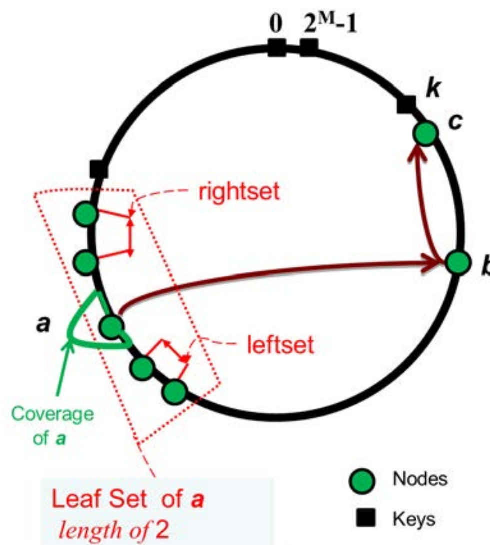

Figure 0.1: L'anneau virtuel de Pastry.

Les principes du fonctionnement de Pastry

Dans Pastry, les nœuds du réseau de recouvrement sont numérotés par des identifiants tirés dans l'intervalle des entiers $[0, 2^M - 1]$ pour un certain M. L'espace des identifiants est considéré comme un anneau, comme c'est montré à la figure 0.1. En particulier, le nœud $2^M - 1$ est voisin du nœud 0.

Les mêmes identifiants servent aussi de clefs d'objets, de manière à ce qu'un nœud gère les clefs qui sont numériquement proches de son identifiant. Il détient ainsi la copie principale de chaque entrée dans la *DHT* associée à l'une de ces clefs. La responsabilité des clefs est répartie uniformément selon la distance entre deux nœuds voisins. Si un nœud est responsable d'une clef nous disons qu'il gère ou qu'il couvre cette clef, comme c'est illustré à la figure 0.1.

Les deux principaux sous-protocoles de Pastry s'appellent *join* et *lookup*. Le protocole *join* permet à un nœud avec un identifiant non encore utilisé de rejoindre l'anneau. Le protocole *lookup* retrouve l'information associée par la *DHT* à une clef donnée (ou, de manière équivalente, le nœud qui couvre cette clef). Le protocole Pastry est censé assurer une association cohérente entre les clefs et les nœuds et fournir un service correct de *lookup* même en présence de *churn*, c'est à dire l'arrivée et le départ spontanés de nœuds.

Comme les voisins d'un nœud dans Pastry peuvent changer dynamiquement à cause de *churn*, chaque nœud maintient dans son état local une structure appelée *leaf set* qui représente son voisinage immédiat. Comme c'est illustré à la figure 0.1, un *leaf set* est constitué de deux ensembles de taille égale indiquant les voisins à gauche et à droite. La taille de ces ensembles est un paramètre de l'algorithme. Le contenu des *leaf sets* est mis à jour lors de l'arrivée de nouveaux nœuds et lorsque le départ d'un nœud est détecté

par un protocole de maintenance. Pour assurer un routage efficace, chaque nœud de Pastry maintient aussi une table de routage qui contient des nœuds plus distants. Dans l'exemple de la figure 0.1, le nœud a reçoit une requête *lookup* pour la clef k. Cette clef n'est pas couverte par a, aussi elle réside en dehors de l'intervalle entre les nœuds les plus distants contenus dans son *leaf set*. La table de routage du nœud a indique le nœud b comme celui dont l'identifiant partage le plus long préfixe commun avec la clef demandée, et a transmet la requête à ce nœud. Le nœud b continue ce processus et enfin, la requête arrive au nœud c qui est le nœud le plus proche à la clef k et qui couvre donc cette clef. On dit que le nœud c *délivre* la requête pour la clef k.

Les résultats de cette thèse

Dans cette thèse nous distinguons les nœuds selon leur statut qui peut être ≪dead ≫, ≪waiting ≫, ≪ok ≫et ≪ready ≫et qui indique son degré d'intégration dans l'anneau et sa capacité à répondre à des requêtes. Seuls les nœuds dont le statut est ≪ready ≫sont supposés avoir une vision cohérente de la couverture des clefs dans leur voisinage et peuvent délivrer des requêtes. Nous définissons le concept de ≪routage fiable ≫par la formule *CorrectDelivery* qui exige qu'à tout moment il existe au plus un nœud qui peut délivrer une requête pour une clef donnée, et qu'il s'agit du nœud ≪ready ≫le plus proche numériquement de la clef. Il n'est pas trivial de garantir cette propriété en présence de *churn*. Nous utilisons une méthode formelle de preuve pour démontrer que la propriété *CorrectDelivery* est vérifiée en présence d'arrivées concurrentes de nouveaux nœuds dans n'importe quelle région de l'anneau, sous l'hypothèse qu'aucun nœud ne quitte le réseau. Nous montrons qu'il est difficile de relâcher cette hypothèse forte car les départs de certains nœuds peuvent induire une séparation permanente du réseau.

0.2 Méthodologie

Les méthodes formelles (Clarke and Wing (1996)) fournissent des techniques systématiques et rigoureuses pour spécifier et vérifier la conception d'un système logiciel. Fondées sur des principes mathématiques et logiques, elles peuvent déceler des problèmes d'inconsistance, d'ambigüité ou d'incomplétude au sein des spécifications d'un système. Avec l'avènement des techniques de vérification algorithmique (*model checking*) et de preuve automatique qui augmentent très significativement le degré d'automatisation d'une preuve interactive, couplées à la haute expressivité de ces méthodes, il est désormais temps d'étudier s'il devient possible d'analyser par des méthodes formelles un protocole réparti d'une complexité réaliste, tel que Pastry.

Questions autour de la méthodologie

Au-delà d'une meilleure compréhension de Pastry, cette thèse illustre comment les méthodes formelles peuvent être utilisées pour l'analyse de systèmes répartis, au-delà de la seule découverte d'erreurs dans des systèmes jouets et abstraits:

- Comment modéliser formellement Pastry? Quel est le niveau d'abstraction adéquat pour la modélisation?
- Comment exprimer formellement les propriétés de correction de Pastry? Est-il possible, et comment, de démontrer leur invariance tout en considérant les détails du protocole, tels que la communication asynchrone et des structures de données complexes?
- À quel degré la vérification formelle d'un tel système peut-elle être automatisée?

Pourquoi TLA$^+$

Cette thèse utilise la méthode TLA$^+$ de Lamport (2002) pour décrire et vérifier la correction du routage et du traitement des requêtes dans Pastry. En effet, TLA$^+$ fournit un cadre logique uniforme pour la spécification, le model checking et la preuve formelle. La structure du langage TLA$^+$ est tres appropriée à la vérification de protocoles car le concept d'actions correspond à la définition de protocoles par des règles en réponse aux messages reçus.

TLA$^+$ est un langage de spécification de haut niveau d'abstraction qui a été utilisé pour spécifier et analyser la correction de plusieurs protocoles matériels et qui est largement utilisé pour la spécification et la vérification d'algorithmes concurrents et distribués. Les notions de TLA$^+$ nécessaires à la compréhension de cette thèse seront introduites en plus de détail dans le chapitre 2.

TLC, introduit dans Yu et al. (1999), est l'outil de vérification algorithmique associé à TLA$^+$. Il est fondé sur des techniques d'exploration explicite de l'espace d'états et permet de détecter des erreurs dans des modèles TLA$^+$. TLC est particulièrement utile pour la validation d'instances de petite taille de modèles TLA$^+$. Il permet de comprendre dans le détail le comportement d'un système décrit par une spécification TLA$^+$ et de découvrir des propriétés et, plus souvent, des non-propriétés, comme cela sera expliqué dans le chapitre 5. TLC peut être exécuté soit à partir d'une ligne de commande sur un serveur, soit à l'aide d'une interface graphique conviviale appelée *Toolbox* qui intègre un éditeur pour le langage de modélisation TLA$^+$, ainsi que les outils d'analyse associés tels que TLC. Une première version de la *Toolbox* TLA$^+$, implantée sous Eclipse[2], est publiquement disponible depuis février 2010 et régulièrement mise à jour depuis.

TLA$^+$ comporte un langage déclaratif de preuve dont la syntaxe et la sémantique sont décrites en détail dans Chaudhuri et al. (2010) et Cousineau et al. (2012). Un exemple d'une preuve en TLA$^+$ apparaît à la fin du chapitre 6, illustrant sa syntaxe et son utilisation pour la preuve de Pastry.

TLAPS (voir Lamport (2012a)) est une plate-forme interactive de preuve qui permet la vérification déductive de propriétés de modèles TLA$^+$. Elle comporte un gestionnaire de preuves (*proof manager*, PM) qui interprète le langage de preuve et déroule les définitions d'opérateurs afin de générer les obligations de preuve correspondantes aux différentes étapes d'une preuve TLA$^+$. Le PM fait ensuite appel à des outils de

[2] `http://www.eclipse.org`

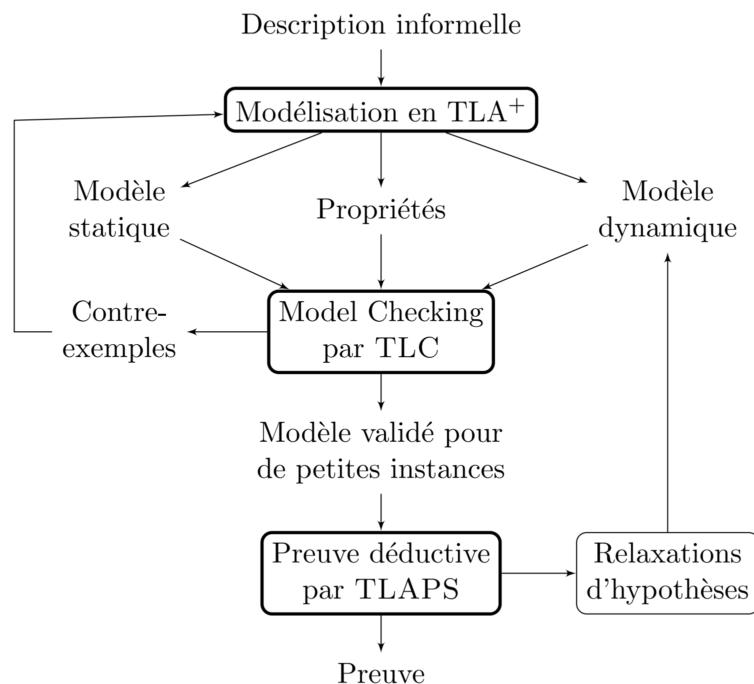

Figure 0.2: Méthodologie de vérification en TLA$^+$.

vérification automatisés pour essayer de démontrer les obligations de preuve générées. Les outils de preuve fournis avec la version initiale de TLAPS ont été Zenon, un prouveur fondé sur la méthode des tableaux (Bonichon et al. (2007)) et Isabelle/TLA$^+$, un encodage de TLA$^+$ en Isabelle/Pure (Wenzel et al. (2008)). Depuis 2012, il existe également une interface avec les outils de résolution SMT (satisfiabilité modulo théories) à partir de TLAPS, tels que Yices (Dutertre and De Moura (2006)), CVC3 (Barrett and Tinelli (2007)) et Z3 (De Moura and Bjørner (2008)).

Comme nous l'avons dit plus haut, le langage TLA$^+$ est bien adapté à la vérification de protocoles parce que son concept d'actions correspond à la définition de protocoles par des règles en réponse aux messages reçus par un nœud du réseau. Aussi, il est aisé de comprendre le langage qui repose sur des concepts élémentaires et classiques mathématiques et logiques. L'intégration de TLC et TLAPS dans la *Toolbox* rend conviviale l'utilisation des outils de model checking et de preuve sur un même modèle. C'est pourquoi nous utilisons TLA$^+$ dans cette thèse pour spécifier un protocole réparti ainsi que pour analyser ses propriétés et pour vérifier sa correction.

Méthodologie générale de vérification en TLA$^+$

La figure 0.2 illustre la méthodologie de vérification en TLA$^+$ qui comporte les étapes de modélisation, de model checking et de preuve déductive.

Partant d'une description informelle d'un système réparti, la première étape est de

encoder les propriétés, les structures de données, le comportement et l'environnement du système par un modèle TLA$^+$. Dans cette thèse nous distinguons différentes formes de modèles TLA$^+$: les *propriétés* expriment des exigences de haut niveau par des formules logiques, le *modèle statique* définit les structures de données par des primitives de TLA$^+$ comme les tableaux, les listes, les fonctions et les enregistrements, et le *modèle dynamique* spécifie le comportement du système par des actions de TLA$^+$. Dans ce qui suit, l'environnement du système correspondra à des hypothèses qui sont formulées et imposées par des pré-conditions d'actions dans le modèle.

L'étape suivante est l'utilisation itérative de TLC pour déboguer et valider le modèle construit. La limitation principale est ici le problème bien connu de l'explosion combinatoire de l'espace d'états, aussi est-il nécessaire de restreindre le modèle à un petit nombre d'instances. Des contre-exemples fournis par TLC servent à analyser et corriger le modèle. Une fois les propriétés validées par TLC (ou au moins que TLC ne découvre plus de contre-exemples après l'avoir laissé tourner pendant suffisamment longtemps), le modèle pourra être vérifié en toute généralité par des preuves TLA$^+$.

La vérification déductive de protocoles distribués repose typiquement sur une preuve inductive qui nécessite la découverte et la formulation d'invariants. TLC est là encore utile pour valider la formulation d'invariants hypothétiques avant d'entamer leur preuve. En général, l'invariant doit être renforcé lors de la construction de la preuve. TLAPS est utilisé pour rédiger et certifier la preuve formelle, en la décomposant en des morceaux suffisamment petits pour que les outils de preuve automatiques arrivent à les vérifier. Le résultat final est une preuve TLA$^+$ dont chaque étape est vérifiée automatiquement par TLAPS.

0.3 Notre preuve de Pastry

Cette section donne un résumé général des défis rencontrés et des leçons que nous avons retenues pendant la préparation de cette thèse.

L'étude de Pastry commence par la modélisation des aspects statiques et dynamiques de l'algorithme. Le modèle formel CastroPastry de Pastry, basé sur Castro et al. (2004), est analysé par le model checker TLC. Des améliorations successives donnent le modèle formel HaeberlenPastry, intégrant des idées décrites dans Haeberlen et al. (2005). Enfin, TLAPS est utilisé pour démontrer que Pastry vérifie bien la propriété *CorrectDelivery* pour un nombre quelconque d'instances. Le protocole Pastry est d'abord vérifié dans sa version IdealPastry et sous deux hypothèse fortes qui sont (1) qu'il n'y a pas deux nœuds qui rejoignent le réseau dans une même région de couverture autour d'un nœud ≪ready ≫et de ses voisins immédiats et (2) qu'aucun nœud ne quitte le réseau. La première hypothèse sera relâchée dans la version LuPastry qui est aussi vérifiée par TLAPS sous l'hypothèse qu'aucun nœud ne quitte le réseau.

Les algorithmes, problèmes et améliorations correspondants aux différentes versions de Pastry seront expliqués en détail dans le chapitre 3. Ci-après nous résumons les principaux défis que nous avons dû confronter au cours de ce travail. Les résultats seront détaillés dans les chapitres ultérieurs de la thèse.

0.3.1 Défis dans la modélisation de CastroPastry et de HaeberlenPastry

Le premier problème a été de déterminer un niveau d'abstraction approprié dans la modélisation formelle de l'algorithme. Par exemple, des bornes temporels servent à limiter les temps d'attente de réponse à un message envoyé. Passé ces délais, un nœud peut supposer que le message a été perdu et soit l'envoyer à nouveau, soit suspecter que le destinataire a quitté le réseau. Afin de simplifier le modèle formel et d'améliorer la tractabilité du problème de model checking, les actions dépendantes du temps réel de Pastry sont représentées en TLA$^+$ comme si elles apparaissaient de manière non-déterministe. Par contre, les détails d'envoi et de réception de messages et le contenu de ces messages ne peuvent être abstraits, et le modèle reflète une communication asynchrone dans laquelle il n'y a pas de garantie quant à la préservation de l'ordre des messages envoyés.

Le second défi a été de compléter des détails qui n'apparaissent pas clairement dans la description de l'algorithme Pastry, à l'aide de contre-exemples fournis par le model checker. Par exemple, il n'est pas dit clairement ce que veut dire qu'une structure locale est ≪complète ≫, une condition nécessaire pour qu'un nœud puisse passer d'une étape à l'autre dans le protocole *join*.

Des hypothèses explicites doivent être prises quant au traitement de cas particuliers, et pour cela nous avons parfois examiné le code source de FreePastry (2009). Ainsi, il peut y avoir un recouvrement des deux *leaf set* d'un nœud, par exemple dans le cas d'un unique nœud actif dans l'anneau. Aucune description particulière de ce cas n'est donnée dans Castro et al. (2004). Nous fournissons une définition précise de la complétude des *leaf set*, et nous décrivons précisément comment sont traités les cas particuliers dans notre modèle. Parfois nous avons établi plusieurs modèles qui décrivent différentes alternatives, fondées sur des hypothèses qui nous paraissaient plausibles.

Un défi dans un autre registre a été de formuler les propriétés de correction car elles ne sont pas indiquées clairement dans Castro et al. (2004). Le concept d'un nœud ayant le statut ≪ready ≫est introduit afin de distinguer les nœuds qui ont une vue locale cohérente de la *DHT*. Seuls ces nœuds peuvent délivrer des requêtes *lookup*. La propriété principale *CorrectDelivery* dit qu'une requête *lookup* pour une certaine clef ne peut être délivrée que par le nœud ≪ready ≫qui est numériquement le plus proche de la clef. Cette propriété est exprimée par une formule temporelle en TLA$^+$.

Le model checker TLC a été utilisé pour valider et améliorer le modèle CastroPastry. Le modèle formel avec quelques résultats du model checking a été publié dans Lu et al. (2010). Le pseudo-code de la version CastroPastry qui contient tous les détails sera donné à la section 3.1.3, et le modèle formel est disponible en ligne à VeriDis (2013).

0.3.2 Analyse des versions CastroPastry et HaeberlenPastry par model checking

Après avoir modélisé Pastry en TLA$^+$ dans le modèle CastroPastry sur la base de Castro et al. (2004), le model checker TLC a été utilisé pour analyser les propriétés de

ce modèle et pour ainsi affiner la compréhension de Pastry.

Un contre-exemple à la propriété *CorrectDelivery* est découvert par TLC pour CastroPastry. Il montre comment deux nœuds peuvent rejoindre le réseau en parallèle entre deux nœuds «ready »sans avoir pris connaissance l'un de l'autre à la fin du protocole *join*. Ce contre-exemple sera décrit en détail à la section 3.2.1.

Il est aisé de voir que ce problème est fondamentalement dû à la communication asynchrone, propriété caractéristique de systèmes distribués qui a pour effet de reordonner les messages par lesquels les nœuds échangent leurs états locaux. Une solution inspirée par Haeberlen et al. (2005) et FreePastry (2009) est d'étendre le protocole *join* par un sous-protocole supplémentaire qui inclut une confirmation de la cohérences des vues du voisinage par un processus appelé «échange de bail », et ce protocole est formalisé par le modèle HaeberlenPastry, expliqué en détail à la section 3.2.2. Ce modèle formel est également disponible en ligne à VeriDis (2013).

De manière analogue à la découverte du contre-exemple pour CastroPastry, plusieurs autres contre-exemples sont découverts automatiquement par le biais des analyses par model checking des versions CastroPastry et HaeberlenPastry, et ils conduisent à des améliorations des modèles de Pastry. Cette analyse du protocole *join* et les améliorations y apportées sont décrites en détail à la section 3.2. Plus de détails sur l'analyse par model checking apparaissent au chapitre 5.

0.3.3 Preuve de réduction

Après que TLC ne trouve plus de contre-exemples lors de l'analyse d'une instance de HaeberlenPastry comportant quatre nœuds au bout de 24h d'exécution, nous sommes assez confiants dans le modèle pour entamer une preuve déductive de correction pour un nombre quelconque de nœuds. Cette preuve est conduite à l'aide du langage de preuve de TLA$^+$, mise en œuvre dans la plate-forme TLAPS pour le développement et la vérification de preuves en TLA$^+$. Un premier résultat publié dans **?** réduit la propriété globale de correction de Pastry appelée *CorrectDelivery* à des invariants sur les structures de données utilisées dans le protocole (voir la section 6.1).

0.3.4 Conception et modélisation de IdealPastry

Une analyse plus poussée du contre-exemple trouvé pour la version CastroPastry conduit à la découverte de la raison fondamentale de la violation de la propriété *CorrectDelivery*: en effet, le protocole choisit d'envoyer l'état du destinataire après sa mise à jour plutôt que celui avant. Alors que le nœud destinataire du message améliore sa vision de l'état global par l'information contenue dans le message, cette mise à jour n'apporte aucune information supplémentaire à l'émetteur du message. Elle peut cependant détruire des informations utiles qui se trouvaient dans l'état local du destinataire avant la mise à jour. Ainsi, une solution à ce problème est de renvoyer l'état antérieur du nœud en réponse au message reçu, et nous allons introduire cette modification au protocole à la section 3.2.4.

De manière générale, l'introduction du protocole «échange de bail »de la version

HaeberlenPastry est nécessaire pour prévenir au problème de nœuds ≪ok ≫qui pourraient être oubliés: un nœud pourrait réussir à échanger des messages *probe* à l'aide de nœuds qui rejoignent l'anneau mais sans être ajouté au voisinage du nœud ≪ready ≫le plus proche. Davantage de détails seront donnés à la section 3.2.4.

Intégrant toutes les améliorations trouvées pendant la phase de model checking, nous obtenons le modèle IdealPastry en TLA$^+$. Ce modèle est disponible en ligne à VeriDis (2013).

Le modèle IdealPastry suppose qu'il n'y a jamais deux nœuds qui cherchent à rejoindre l'anneau entre deux mêmes nœuds ≪ready ≫proches l'un de l'autre. Cette hypothèse est réalisée en empêchant un nœud à traiter d'autres requêtes *join* dès lors qu'il répond à une requête *join* d'un nœud proche. Au-delà de ce nœud, ses voisins doivent également être empêchés à traiter des requêtes *join*. Puisque le fait de bloquer d'autres nœuds nécessite une communication entre les nœuds, la version IdealPastry ne décrit qu'un protocole abstrait et idéalisé qui nécessite une réalisation algorithmique pour sa mise en œuvre.

0.3.5 Validation de IdealPastry

Le modèle formel IdealPastry est validé à l'aide de TLC, et le résultat peut être trouvé en ligne au même endroit que le modèle à VeriDis (2013).

0.3.6 Vérification de IdealPastry

Le modèle IdealPastry est vérifié par une preuve inductive d'invariants, sous la double hypothèse qu'aucun nœud ne quitte le réseau et qu'il n'y ait pas deux nœuds qui cherchent à rejoindre l'anneau dans la région couverte par un nœud ≪ready ≫et ses voisins immédiats.

La partie la plus subtile de cette preuve déductive est la définition d'un invariant inductif approprié qui implique la propriété qui nous intéresse et qui est préservé par toutes les actions du protocole. Cette définition est guidée par construction de la preuve, lors de laquelle nous utilisons TLC pour découvrir tôt des violations inattendues d'invariants hypothétiques. Plus précisément, les défis suivants sont confrontés pendant la preuve d'invariants inductifs.

D'abord, il faut séparer consciencieusement les lemmes ≪statiques ≫sur les structures de données des invariants sur le comportement dynamique du système. Puisque le modèle qui représente Pastry contient plusieurs structures de données complexes telles que l'anneau, les *leaf set* et la table de routage avec des opérations complexes, les propriétés de ces structures de données et des opérations associées sont démontrées séparément de la preuve inductive, afin que cette dernière soit focalisée sur l'aspect dynamique. Dans la syntaxe des formules, cette séparation est indiquée par le fait qu'une formule porte sur toute instance arbitraire de la structure de données ou qu'elle concerne une variable d'état précise qui implante la structure en question dans le protocole. Par exemple, une propriété que nous avons eu à démontrer énonce que l'ajout d'un nœud à un *leaf set* appartenant à un nœud n ne peut que réduire la distance entre n et ses

voisins immédiats. Bien que cette propriété évoque la modification d'un *leaf set*, elle s'applique à tout nœud et non seulement à un nœud particulier dans une certaine phase du protocole. Il s'agit donc d'une propriété statique concernant la structure de données des *leaf set* et non d'un invariant. Une autre propriété dit que si le nœud i appartient au *leaf set* d'un autre nœud j qui n'est pas en train d'aider i à rejoindre l'anneau, alors le *leaf set* du nœud i n'est pas vide. Cette propriété est un invariant car la question si le *leaf set* du nœud i est ou n'est pas vide est une propriété dynamique.

Ensuite, il peut être difficile de trouver des généralisations appropriées d'invariants qui doivent être démontrés à être préservés par toutes les actions du système de transitions correspondant à l'algorithme Pastry. Effectivement, l'ensemble des invariants ne peut être finalisé qu'en même temps que les preuves car un nouvel invariant pourra être découvert lors de la preuve d'un cas particulier d'un invariant existant. Il est important d'esquisser la preuve entière d'invariance a priori afin d'éviter d'être perdu dans les détails de formulation. L'attention à des cas particuliers et la découverte de contre-exemples sont utiles pour la construction de l'invariant. Il n'est pas toujours possible d'utiliser le model checker: un cas particulier important n'a été trouvé qu'à la main pendant la recherche d'invariants. Ce contre-exemple à la propriété *CorrectDelivery* n'apparaît que si plus de cinq nœuds interviennent, dont trois qui rejoignent l'anneau à des positions spécifiques. Les détails seront expliqués à la section 6.4.2.

Troisièmement, il n'est pas toujours trivial de donner les arguments appropriés à la preuve mécanique, y compris les appels nécessaires à des lemmes et faits précédemment démontrés. Des étapes ≪évidentes ≫pour l'humain qui applique un raisonnement logique implicite peuvent devenir des étapes compliquées de preuve nécessitant l'énumération de lemmes triviaux mais nécessaires. Parfois l'esquisse manuelle de la preuve comporte effectivement des lacunes importantes qui doivent être adressées. Dans ce cas il faut construire une nouvelle esquisse de preuve, nécessitant sa reconstruction ou parfois même une modification de l'invariant. C'est pourquoi le recours à un outil de preuve automatique est une aide cruciale à la construction et la vérification de la preuve. Dans d'autres cas, l'outil peut être incapable de démontrer une certaine obligation pourtant valide, et il faut alors décomposer sa preuve en un niveau supplémentaire d'interaction.

À la fin, une preuve déductive complète a été construite dans le langage de preuve de TLA$^+$. Elle consiste en environ 10000 étapes de preuve qui sont toutes vérifiées à temps par TLAPS en faisant appel à différents outils de preuve automatisés. Les détails seront expliqués à la section 6.3.

0.3.7 Relâcher les hypothèses

Un contre-exemple à la propriété *CorrectDelivery* sur le modèle IDEALPASTRY est découvert à la main si l'hypothèse sur les nœuds cherchant à rejoindre l'anneau en parallèle dans une région circonscrite est relâchée. Plus précisément, lorsque trois ou plus nœuds rejoignent l'anneau simultanément à des endroits précis, un nouveau nœud ≪ready ≫peut être ignoré par des nœuds ≪ready ≫proches existants. Les détails seront expliqués à la section 6.4.2.

0.3.8 Conception et modélisation de LuPastry

L'analyse du problème de requêtes *join* concurrentes conduit à une amélioration de la conception de Pastry appelé LuPastry. Dans cette version, un nœud «ready »ajoute directement le nœud qui cherche à rejoindre l'anneau dès la réception de la requête, et n'accepte aucune nouvelle requête *join* avant d'avoir reçu la confirmation que le nouveau nœud est lui-même devenu «ready ». Le modèle formel de LuPastry est expliqué au chapitre 4.

0.3.9 Validation de LuPastry

Le modèle LuPastry est validé par TLC, et les résultats apparaissent au tableau 5.3.

0.3.10 Vérification de LuPastry

Nous démontrons que le modèle LuPastry vérifie également la propriété *CorrectDelivery* pour peu qu'aucun nœud ne quitte le réseau. Les invariants de ce modèle sont introduits à la section 6.5.

Il n'est pas trivial d'adapter la preuve de IdealPastry à LuPastry. TLAPS dispose d'une commande pour examiner le statut d'une preuve et de ses sous-preuves, et cette commande indique quelles étapes de la preuve sont affectées par les modifications du modèle. Cependant, de nombreuses étapes de preuve ne sont plus valides, et ces étapes doivent être étudiées attentivement. Les détails des modifications apportées aux invariants et aux preuves sont indiqués à la section 6.4.4.

Un exemple d'une preuve en TLA^+ est donné à la fin du chapitre 6, expliquant la syntaxe et l'usage de TLAPS. La preuve formelle entière est trop longue pour être incluse dans cette thèse; elle est disponible en ligne à VeriDis (2013).

0.4 Contributions de la thèse

La contribution principale de cette thèse est l'analyse et l'amélioration d'un algorithme réel Pastry réalisant une *DHT*. Ainsi, nous illustrons et étendons l'usage de méthodes formelles pour la vérification d'algorithmes répartis. Cette section résume nos contributions qui ont en fait déjà été mentionnées dans les sections précédentes.

- L'algorithme Pastry introduit par Castro et al. (2004) est modélisé formellement au niveau des échanges de messages par le modèle CastroPastry (section 3.1.3). Il s'agit du premier modèle formel et complet d'un tel algorithme complexe *P2P*. Sa version améliorée par Haeberlen et al. (2005) est également représentée dans le modèle HaeberlenPastry.

- Une propriété fondamentale de correction de Pastry est définie et représentée par la formule *CorrectDelivery* (section 4.3.2). Elle dit qu'à tout moment il existe au plus un nœud qui peut délivrer une clef, et que c'est le nœud «ready »le plus proche de la clef.

- Le protocole *join* de Pastry est étudié en détail. Des violations de *CorrectDelivery* par les modèles CastroPastry et HaeberlenPastry sont exhibées et analysées aux sections 3.2 et 5.2.

- Les modèles HaeberlenPastry, IdealPastry et LuPastry introduisent différents statuts de nœuds (≪dead ≫, ≪waiting ≫, ≪ok ≫et ≪ready ≫) reflétant leurs progressions dans le protocole *join* et leurs capacités à délivrer des clefs (section 3.3.2). Les seuls nœuds ≪ready ≫sont supposés de disposer d'informations cohérentes quant à la couverture de clefs et sont autorisés à délivrer des requêtes *lookup*.

- Le modèle IdealPastry résume les améliorations induites par l'analyse de CastroPastry et HaeberlenPastry. Nous démontrons par une preuve inductive d'invariants qu'il vérifie la propriété *CorrectDelivery*, sous les hypothèses qu'aucun nœud ne quitte le réseau et que deux nœuds ne cherchent jamais à rejoindre l'anneau dans la région couverte par un nœud ≪ready ≫et ses voisins immédiats.

- Une spécification formelle LuPastry de l'algorithme Pastry est conçue sur la base de IdealPastry (introduite en pseudo-code à la section 3.3.3). Son amélioration réside dans le fait qu'un nœud ≪ready ≫ajoute un nœud arrivant dans le réseau dès qu'il traite la requête *join* et n'accepte alors plus de requête *join* jusqu'à ce qu'il reçoive la confirmation que le nouveau nœud est devenu ≪ready ≫. Le modèle LuPastry est démontrée correct par rapport à *CorrectDelivery* sous l'hypothèse qu'aucun nœud ne quitte l'anneau. Les invariants utilisés dans la preuve sont introduits à la section 6.5. Il n'est pas facile de relâcher cette hypothèse sur le départ de nœuds à cause de contre-exemples qui montrent qu'une séparation permanente du réseau peut intervenir lorsque plusieurs nœuds partent simultanément.

- Le model checker TLC est utilisé de manière intensive pour déboguer et valider les différents modèles de Pastry sur des instances impliquant entre trois et cinq nœuds. Des versions simplifiées sont conçues afin de surmonter le problème de l'explosion combinatoire de l'espace d'états. Nous résumons nos expériences et nos propositions quant à l'utilisation de TLC au chapitre 5 qui donne également les détails de nos résultats.

- La plate-forme TLAPS de preuve interactive est utilisée afin de conduire les preuves formelles démontrant que les modèles IdealPastry et LuPastry vérifient la propriété *CorrectDelivery*. Cette preuve montre la correction des modèles formels quant à la propriété visée; elle illustre également la possibilité d'utiliser TLAPS pour une preuve de grande taille impliquant plus de 10000 étapes de preuve.

- L'étude de cas qui consiste à démontrer que les modèles IdealPastry et LuPastry vérifient *CorrectDelivery* met en évidence la possibilité de conduire de manière interactive la preuve de correction d'un système de transition complexe et indique la démarche à suivre. Le chapitre 6 explique cette méthodologie en démontrant étape par étape à quel degré les hypothèses sont vérifiées et comment la preuve peut être complétée.

0.5 Structure du mémoire de thèse

La suite de ce mémoire est structurée comme suit. Le chapitre 2 donne un état de l'art concernant les systèmes *P2P* et les techniques de vérification formelle nécessaire à la compréhension de ce qui suit.

Le chapitre 3 introduit le protocole Pastry au niveau conceptuel, partant du pseudo-code pour CastroPastry, fondé sur Castro et al. (2004), en passant par des diagrammes de flot pour HaeberlenPastry, inspiré par Haeberlen et al. (2005) et IdealPastry en tant que premier modèle formellement vérifié, et aboutissant à LuPastry, introduit à nouveau par du pseudo-code et des éléments de sa preuve.

Au chapitre 4 nous introduisons le modèle formel de LuPastry et ses propriétés de correction. Le chapitre 5 illustre l'utilisation du model checker TLC pour la validation des modèles et l'analyse de la propriété de correction. Quelques contre-exemples illustratifs sont présentés en détail. La démarche de la preuve déductive est introduite au chapitre 6 et appliquée à la vérification de HaeberlenPastry, IdealPastry et LuPastry.

Le chapitre 7 introduit d'autres systèmes *P2P* réalisant des *DHT* et compare le travail mené dans cette thèse avec d'autres approches pour la vérification de protocoles de réseaux.

Au chapitre 8 nous résumons cette thèse en rappelant ses contributions, et nous énonçons quelques perspectives pour des travaux futurs.

1 Introduction

Pastry (Rowstron and Druschel (2001), Castro et al. (2004), Haeberlen et al. (2005)) is a structured *P2P* algorithm realizing a Distributed Hash Table (*DHT*, by Hellerstein (2003)) over an underlying virtual ring of nodes. Several implementations of Pastry are available and have been applied in practice, but no attempt has so far been made to formally describe the algorithm or to verify its properties. Since Pastry is a typical realization of a *DHT* which combines rather complex data structures, asynchronous communication, concurrency, resilience to churn, i.e. concurrent join and departure of nodes, it makes an interesting target for verification.

This thesis models Pastry's core algorithms, which provide the correct lookup service in the presence of churn and maintain a local data structures to adapt the dynamic updates of neighborhood. This chapter starts with the motivation of the research interests, then states explicitly the research goals and explains how these goals are achieved by the work. At the end, a structural guidance will be given for reading the thesis.

1.1 Motivation

In our day-to-day life, disruptions of software systems such as SKYPE (Microsoft (2013)) directly disturb everyone's daily life when people cannot make calls, or calls are dropped in the middle of a conversation. SKYPE is known and used world-wide by billions of users for making phone-calls over Internet. On Thursday, 16th August 2007, the SKYPE peer-to-peer (*P2P*) network became unstable and suffered a critical disruption despite of its peer-to-peer network with an inbuilt ability to self-heal. "This event revealed a previously unseen software bug within the network resource allocation algorithm which prevented the self-healing function from working quickly." as reported in Arak (2007). Unfortunately on 23rd December 2010, millions of users could not make phone calls again due to the departure of several "super nodes". Is there any fundamental problems of such network design that causes unexpected outage again and again? If not, is there a fundamental proof of its correctness?

Although it is not published how SKYPE uses the idea of *P2P* algorithm in its deployments of services, there is no doubt that the scalability of SKYPE benefits significantly from the adoption of *P2P* networks, making it feasible to provide billions of phone calls simultaneously world-wide, according to Microsoft (2013). *P2P* systems have become popular since beginning of 21st century with their self-organization and decentralization properties. In particular, structured *P2P* systems implement Distributed Hash Tables (*DHTs*), which are supposed to provide

- efficient and dependable routes;

- distributed maintenance of structure with low costs;
- and resilience to concurrent joins and departures of network members.

For these reasons, *DHT* is typically used in large-scale distributed systems such as Dynamo (DeCandia et al. (2007)), a storage substrate that Amazon uses internally for many services. SKYPE may implement *DHT* for its peers to find each other correctly and efficiently.

However, the correctness of a *DHT* system relies heavily on its realizing algorithms. Pastry is one of the successful realization[1] of *DHT*. It realizes *DHT* by mapping object keys (e.g. identifier for a piece of distributed data) to overlay nodes (e.g. the computer connected to the Internet) and offers a *lookup* primitive to route a message to the node responsible for a key.

Questions About Pastry

Since Pastry implements all properties of a *DHT*, it is interesting and crucial to understand the fundamental mechanisms of Pastry and to analyze and prove its correctness properties. Castro et al. (2004) introduces Pastry on the message level using pseudocode. Starting from this paper, it will be interesting to know the insights of Pastry:

- How does Pastry work, in particular, how does the protocol realized the *DHT*?
- What does "dependable routing" formally mean? Does Pastry guarantee dependable routing? If so, to what extent, and how? Is there any fundamental design flaw in Pastry, in particular Castro et al. (2004), with respect to its correctness properties, such as dependable routing? If yes, how does the problem occur? If not, is there a formal proof?
- Are there other interesting properties of such a system implementing a *DHT*? Are they interrelated?

How does Pastry Work

In Pastry, the overlay nodes are assigned logical identifiers from an Id space of naturals in the interval $[0, 2^M - 1]$ for some M. The Id space is considered as a ring as shown in Figure 1.1, i.e. $2^M - 1$ is the neighbor of 0.

The Ids are also used as object keys, such that an overlay node is in particular responsible for keys that are numerically close to its Id, i.e. it provides the primary storage for the hash table entries associated with these keys. Key responsibility is divided equally according to the distance between two neighbor nodes. If a node is responsible for a key we say it *covers* the key, as illustrated in Figure 1.1.

[1] This thesis distinguishes the level of abstraction of an algorithm. Therefore, "realization" is used to described the refinement from *DHT* to a real algorithm Pastry and "implementation" is used to describe different versions of this algorithm. Both of the words describe different level of details of design and neither of these two words are used in the meaning of executable software.

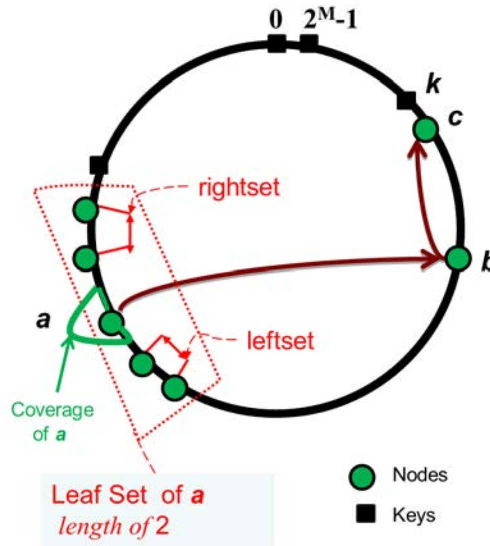

Figure 1.1: Pastry ring.

The most important sub-protocols of Pastry are *join* and *lookup*. The join protocol eventually adds a new node with an unused network Id to the ring. The lookup protocol delivers the hash table entry (the responsible node) for a given key. Pastry maintains its correct key mapping and it is supposed to provide the correct lookup service in the presence of *churn*, i.e. spontaneous join and departure of nodes.

Since neighbors of a Pastry node are changing all the time due to churn, they are locally decided based on the *leaf sets* of the node, a local state that each Pastry node maintains. Leaf sets, as shown in Figure 1.1, consist of a left set and a right set of the same length which is a parameter of the algorithm. The nodes in leaf sets are updated when new nodes join or failed nodes are detected using maintenance protocol. In order to achieve efficient routing, a Pastry node also maintains a routing table to store more distant nodes. In the example of Figure 1.1, node a received a lookup message for key k. The key is outside node a's coverage and furthermore, it doesn't lie between the leftmost node and the rightmost node of its leaf sets. Querying its routing table, node a finds node b, whose identifier matches the longest prefix with the destination key and then forwards the message to that node. Node b repeats the process and finally, the lookup message is answered by node c, which is the closest node to the key k, i.e. it covers key k. In this case, we say that node c *delivers* the lookup request for key k.

Result of This Thesis

This thesis separates different statuses ("dead", "waiting", "ok" and "ready") of an overlay node according to its stage during join and readiness for delivering a key. Only "ready" nodes are supposed to have consistent key mapping among each other and are allowed to deliver a message. This thesis defines the "dependable routing" as the

correctness property *CorrectDelivery*, requiring that there is always at most one node that can deliver a lookup request for a key. This property is non-trivial to obtain in the presence of *churn*. Using formal method, this thesis proves that *Correct Key Delivery* holds for all “ready” nodes under assumption that no nodes leave the network, despite concurrent joins of new nodes in arbitrary region. The assumption is not further relaxed due to possible separation of network problem caused by particular departure of nodes.

1.2 Methodology

Formal methods (Clarke and Wing (1996)) provide a systematic and rigorous way to specify and verify the design of a software system. Using mathematics and logic expressions, they can reveal inconsistencies, ambiguities and incompleteness that are often undetected by people writing the requirements. With the development of model checking and automated theorem proving techniques which then improved the interactive theorem proving method, it is now the time to benefit from their improved efficiency and expressive power to see if it is now feasible to formally analyze a complex real world distributed system such as Pastry.

Questions About Methodology

Besides the deeper understanding of Pastry, this thesis illustrates also how formal methods can be applied to analyze distributed systems beyond only finding bugs in abstract toy examples:

- How to formally model Pastry? What is a proper abstract level of modeling?
- How to express the correctness properties of Pastry in TLA$^+$? Is it at all feasible to prove their invariance considering the intricacies of message interleaving and complex data structures of Pastry, and how?
- To what extent can the formal verification of such a system be conducted automatically?

Why TLA$^+$

This thesis employs TLA$^+$ by Lamport (2002) to analyze and verify the correct delivering and routing functionality of Pastry, because TLA$^+$ provides a uniform logic framework for specification, model-checking and theorem proving. Besides, it fits protocol verification quite nicely, because its concept of actions matches the rule/message-based definition of protocols.

TLA$^+$ by Lamport (2002) is a high-level specification language that has been used to specify and check the correctness of several hardware protocols, it is now widely used for specifying and verifying concurrent and distributed algorithms. The necessary foundations of TLA$^+$ are introduced in Chapter 2.

TLC, introduced in Yu et al. (1999) is the model checker implemented for TLA$^+$ using explicit state exploration techniques to help finding errors in a TLA$^+$ specification. TLC is in particular helpful for validating TLA$^+$ models for small finite instances, for understanding the detailed behavior of the system modeled using TLA$^+$, and for discovering properties and, more frequently, non-properties as explained later in Chapter 5. TLC can be launched either by command line on a server or within a user-friendly IDE, the Toolbox, which integrate an editor for writing TLA$^+$ and relevant languages for modeling algorithm and its tools such as TLC to debug the model. A first version of Toolbox for TLA$^+$ based on Eclipse[2] was released in February 2010.

TLA$^+$ contains a declarative language for writing hierarchical proofs. Its syntax and semantics are explained in detail in Chaudhuri et al. (2010) and Cousineau et al. (2012). An example of a proof in TLA$^+$ is illustrated at the end of Chapter 6 to demonstrate its syntax and usage for proving Pastry.

TLAPS in Lamport (2012a) is an interactive theorem proving system for deductively verifying properties of TLA$^+$ specifications and automatically checking TLA$^+$ proofs. It is built around a Proof Manager (PM) to interpret this proof language, to expand the necessary module and operator definitions, to generate corresponding proof obligations, and to pass them to back-end automated verifiers. The initially supported back-ends are the tableau prover ZENON (Bonichon et al. (2007)) and ISABELLE/TLA$^+$, an axiomatization of TLA$^+$ in ISABELLE/PURE (Wenzel et al. (2008)). Since 2012, SMT back-end is also available by TLAPS to launch SMT solvers, such as Yices (Dutertre and De Moura (2006)), CVC3 (Barrett and Tinelli (2007)) and Z3 (De Moura and Bjørner (2008)).

TLA$^+$ fits protocol verification quite nicely, because its concept of actions matches the rule/message-based definition of protocols. Besides, the specification language is straightforward to understand with basic mathematics and classical first-order-logic. Furthermore, the convenient toolbox available in Lamport (2012a) includes now both the TLC model checker and the TLAPS PM. Therefore, this thesis uses TLA$^+$ to specify a distributed system, to analyze its properties and to verify its correctness.

The TLA$^+$ Framework for Verification

Figure 1.2 illustrates the complete process of this framework which includes modeling, model checking and theorem proving.

Starting with an informal description of a distributed system, the first task is to model the requirements, data structures, behavior and environment of the system to the TLA$^+$ model. This thesis distinguishes different kinds of TLA$^+$ model as *properties* that specifies requirements using logic formula, the *static model* that defines the data structures using the TLA$^+$ primitives such as arrays, lists, functions and records, and the *dynamic model* that describes the behavior using actions of TLA$^+$. In this thesis, environment of the system corresponds to assumptions; they are formulated and implemented as preconditions of the actions in the model.

[2] www.eclipse.org

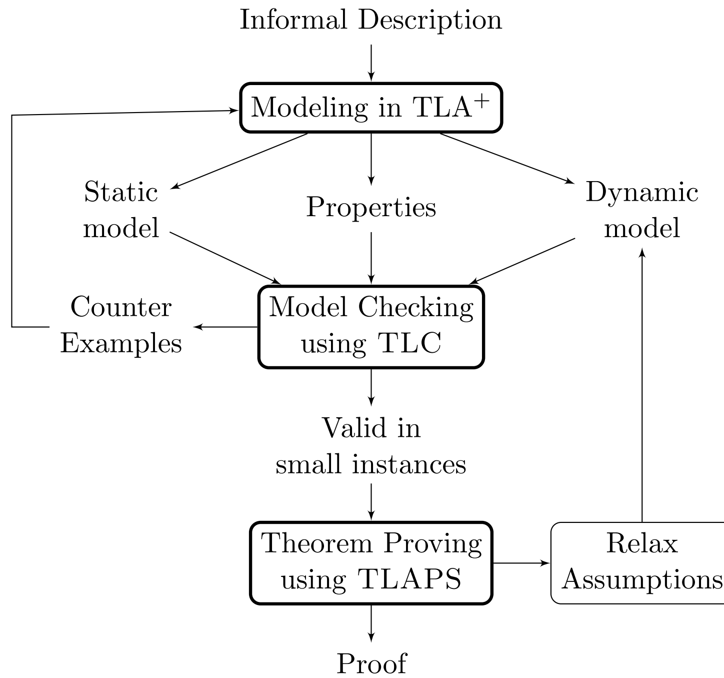

Figure 1.2: Verification approach using TLA$^+$.

Using TLC to debug and validate this model is an iterative process coming as the next step. Model analysis helps to discover unexpected corner cases to improve the model and validation ensures that the system has at least some useful executions, e.g., accessibility properties are model-checked by checking that the negation is false. Note that only a restricted number of instances can be verified using TLC in order to avoid the state explosion problem. By counterexamples, the model will be analyzed and reformulated. By successful validation or no counterexamples after running it for considerably long time, the model will then be verified by TLA$^+$ proofs.

The proof for distributed systems typically contains an induction part, where invariants need to be found and formulated. The model checker can be applied again to debug the formulation errors or discover invalid hypothetical invariants in a early stage. Usually the invariant is extended during the process of being proved. TLAPS is used to write the proof manually and sometimes break it down into small enough pieces so that it can be checked automatically using back-end prover. The final verification result is a TLA$^+$ proof, of which each proof step is automatically verified using TLAPS.

1.3 The Story of Verifying Pastry

This section gives a general overview of the challenges encountered and lessons learned throughout this thesis.

The analysis of Pastry starts with modeling the static and dynamic behavior of Pastry. The formal model of Pastry, CastroPastry based on Castro et al. (2004), is analyzed using model checker TLC and improved to the formal model HaeberlenPastry inspired by Haeberlen et al. (2005). In order to verify that Pastry conforms *CorrectDelivery* in arbitrary number of instances, TLAPS is used for theorem proving approach. The Pastry protocol is first verified in version IdealPastry with two strong assumptions that (1) no concurrent joins occur in the coverage regions among a "ready" node and its neighbors and (2) no nodes leave the network. The assumptions are relaxed by version LuPastry to the assumption that no nodes leave the network. Based this assumption, LuPastry is also verified using TLAPS. All the different versions of Pastry are validated to be able to perform successful lookup and join.

The algorithms, problems and improvements of Pastry are explained in detail in Chapter 3. Different challenges are confronted during these approaches, which are summarized here as a story of verifying Pastry. The results are then unfolded in later Chapters of the thesis in detail.

1.3.1 Challenges of Modeling CastroPastry and HaeberlenPastry

The first challenge was to determine an appropriate level of abstraction. For example, time thresholds are used to restrict the waiting time for a reply in a network protocol such as Pastry, so that a node can assume the loss of message and repeat the request or suspect the other side to be faulty. In order to simplify the formal model for tractability of model checking, timing-dependent actions are represented in TLA$^+$ as occurring non-deterministically. However, details of sending and receiving messages together with the necessary message content cannot be abstracted away due to the analysis on the effect of all possible message interleaving in a network.

The second challenge was to fill in unstated details in the description of the Pastry algorithm, based on the counterexamples revealed by model checking. For instance, it was not explicitly stated what it means for a local data structure to be "complete", which is crucial for a node to decide if it can proceed to the next stage of join.

Explicit assumptions are made on how corner cases should be handled, sometimes based on an exploration of the source code of FreePastry (2009). For example, the leaf sets of a node can be overlapping and not complete if there is only one node on the network. No specific description can be found in Castro et al. (2004). Thus, precise definition of completeness of leaf sets is formally defined together with rigorous description of handling the corner cases. In other cases, different models are established describing alternatives based on different assumptions that appeared reasonable.

A further challenge was to formulate the correctness properties themselves because they are not explicitly stated in Castro et al. (2004). A notion of nodes being "ready" is introduced to distinguish nodes with a consistent local view of the distributed hash table from those without. Therefore, only "ready" nodes handle lookup messages. The mainly focused correctness property *CorrectDelivery* states that the lookup message for a particular key is always answered by the numerically closest "ready" node. This property is expressed as a temporal formula in TLA$^+$.

The model checker TLC was employed to help debug and improve the model CastroPastry. The formal model with some model checking examples are published in Lu et al. (2010). The pseudocode of CastroPastry with filled details is illustrated in Section 3.1.3 and its formal model is available online at VeriDis (2013).

1.3.2 Model Checking CastroPastry and HaeberlenPastry

After modeling Pastry in TLA$^+$ as CastroPastry based on Castro et al. (2004), the model checker TLC is applied to analyze the properties of it, which gives more insights of Pastry.

A counterexample of CastroPastry is discovered by TLC which violates the desired property *CorrectDelivery*. In that case, two nodes joining concurrently between two "ready" nodes fail to know each other by the end of the join process. More details of the counterexample can be found in Section 3.2.1.

An obvious reason is the interleaving of the messages exchanging local states of nodes, which is inevitable for a distributed system. A solution inspired by Haeberlen et al. (2005) and FreePastry (2009) is to extend the join protocol with further subprotocol that includes a confirmation of neighborhood by lease granting process, which leads to the Pastry model HaeberlenPastry, explained in detail in Section 3.2.2. The formal model of HaeberlenPastry is available online at VeriDis (2013).

Similar to the counterexample of CastroPastry discussed above, several other counterexamples are discovered automatically through model checking CastroPastry and HaeberlenPastry using TLC, which guides the improvements of the Pastry models. The analysis and improvements of join protocol of Pastry is explained in Section 3.2. Further details of model checking analysis are shown in Chapter 5.

1.3.3 Reduction Proof

After the model checker cannot find any more counterexamples on analyzing HaeberlenPastry with four nodes for a day, enough confidence is gained to move on to see if the correctness can be proved for arbitrary number of nodes thoroughly. This is achieved by theorem proving using the proof language of TLA$^+$ and the interactive theorem prover TLAPS, a platform for the development and verifications of TLA$^+$ proofs. A first result published in **?** reduces the global Pastry correctness property *CorrectDelivery* to the invariants of the underlying data structures of Pastry (Section 6.1).

1.3.4 Design and Modeling IdealPastry

Further analysis of the counterexample of CastroPastry leads to the discovery of the crucial reason for the violation discussed above: the design decision of exchanging updated information rather than "outdated" one. Since a node only improves its local state with the knowledge from the received message during an action, the update of this local state does not bring new information back to the sender of the message, but may erase some useful one previously stored there. Hence, a solution to this problem is to

send back the previous state of the node in reply to the received message, introduced in Section 3.2.4.

Anyway, the lease granting protocol of HaeberlenPastry has been shown to be necessary to prevent the problem of ignored "ok" nodes, where node completes the probing process with help of joining nodes, but without being added into the closest "ready" node. More details are illustrated in Section 3.2.4.

Combining all improvements from model checking approach, the formal model IdealPastry is designed and modeled in TLA$^+$. The formal model is available online at VeriDis (2013).

IdealPastry assumes no concurrent join between two "ready" nodes close to each other. This assumption is realized by blocking the node from receiving further join request when it starts handling a join requests from nearby node. The blocking node also "block" its neighbors such that no join requests can be handled by them either. Since blocking other nodes needs network communication, this version describes only an abstract and ideal protocol, which needs to solve further problem by realization.

1.3.5 Validation of IdealPastry

The formal model IdealPastry is validated using TLC and the result can be found on the Web together with the model in VeriDis (2013). Here the validation ensures that the system has at least some useful executions. For instance, accessibility properties are model-checked by checking that the negation is false.

1.3.6 Verification of IdealPastry

IdealPastry is verified through the inductive proof of invariants, under the assumption that no nodes leave the network and no concurrent join occur within the coverage region of a "ready" node and its direct neighbors.

The most subtle part of the verification approach is the search for an appropriate inductive invariant which implies the required property and is inductively preserved by all possible actions. The search is guided by the construction of the proof, where TLC is used to discover the unexpected violations of hypothetical invariant in a earlier stage.

More precisely, the following challenges are confronted during the proof of inductive invariants.

Firstly, Lemmas on the static data structure must be carefully separated from invariants of the dynamic behavior of the system. Since the model that Pastry implements contains many complex data structures such as ring, leaf set and routing table with complicated operators, properties of these data structures and operators are proved aside from the induction proof, in order to leave the induction proof with more focus on the dynamic aspect. The separation relies on checking if a formula only talks about an instance of data structure or about a state variable, which implements a data structure. For example, a property states that adding a new node into a leaf set can only make the direct neighbor of that node become closer to it. Although this property talks about the modification of a leaf set, it does not talk about the leaf set of a particular node in a

state, therefore, this is only a lemma about the leaf set data structure, not an invariant. Another property states that if a node i is a member of the leaf set of another node j, which is not currently helping i to join, then the leaf set of the node i is not empty. This property is then an invariant, because the emptiness of a leaf set of a particular node is a dynamic property.

Secondly, it is subtle to find the appropriate generalization of invariants, which needs to be inductively proved throughout the transition system modeling Pastry algorithm. In fact, the set of invariants can only be finalized together with their proofs, because new ones might be discovered by proving some corner case of one particular invariant. A proof sketch for the complete invariant proof must be conducted a priori to make the global picture clear, such that one does not get lost in the formulation details. Corner cases and counterexamples may help to construct the invariant. Using model checker is not enough, because one important corner case was discovered manually during the search for invariants. This counterexample violating the correctness property *CorrectDelivery* only occurs when more than five nodes are involved and three of them are concurrently joining with specific positions on the ring. More details are explained in Section 6.4.2.

Thirdly, it is not trivial to find the appropriate proof arguments, including necessary lemmas and previous proof results. Many "obvious" steps with implicit logical reasoning for a human become a complicated proof with explicit enumeration of trivial but necessary lemmas. Sometimes, there are indeed important flaws in proof sketch that need to be taken care of. In this case, new proof sketch needs to be constructed. There can be even errors in the proof, which lead to reconstruction of the proof, or even modification of the invariant. For this reason, an automated back-end theorem prover plays a crucial role to aid the construction and verification of the proof. The prover can sometimes be unable to prove certain obligations, which must then be broken down in an additional level of interaction.

In the end, a complete proof was constructed in TLA$^+$ proof language consisting of about 10000 proof steps, which are interactively checked in time by using TLAPS launching different back-end automated theorem provers. More details are explained in Section 6.3.

1.3.7 Relaxing the Assumptions

A counterexample of *CorrectDelivery* of IdealPastry is discovered manually when the assumption of concurrent join is relaxed. More precisely, when three or more nodes with particular identifiers join simultaneously into the network, a newly joined "ready" node may be ignored by the nearby "ready" nodes. More details are explained in Section 6.4.2.

The assumption that no nodes leave the network cannot be relaxed due to network separation when particular nodes in the important position in the network departs simultaneously. More details are explained in Section 6.4.1.

1.3.8 Design and Modeling of LuPastry

The analysis of the concurrent join problem leads to an improved design of Pastry, LuPastry, in which a "ready" node adds the joining node directly when it receives the join request and the "ready" node does not accept any new join request until it gets the confirmation that the current joining node is "ready". The formal model of LuPastry is shown in Chapter 4.

1.3.9 Validation of LuPastry

LuPastry is validated using TLC and the results are summarized in Table 5.3.

1.3.10 Verification of LuPastry

LuPastry is also verified to conform to the desired property *CorrectDelivery* under the assumption that no nodes leave the network. Its invariants are introduced in Section 6.5.

It is not easy to adapt the proof from IdealPastry to LuPastry. TLAPS has a command for status checking, which shows which steps have been broken by a change. However, TLAPS cannot automatically recheck the proof by one click. It reports failures at many places, which then need to be carefully analyzed. Details of modification of the Proof and Invariants are explained in Section 6.4.4.

An example of proof in TLA^+ is illustrated at the end of Chapter 6, in order to explain its syntax and usage. The complete formal proof is too large to be included in the thesis and therefore put on the Web in VeriDis (2013).

1.4 Contributions of the Thesis

This thesis has mainly contributed in analyzing and improving a real world *DHT* algorithm Pastry, and demonstrating and extending the usage of formal methods of verification of distributed algorithms. This section summarizes these contributions which are in fact all mentioned in the previous sections.

- The Pastry algorithm of Castro et al. (2004) is formally modeled on the message exchange level in CastroPastry (Section 3.1.3), which is the first formal model of such a complex real world *P2P* algorithm. Its improved version of Haeberlen et al. (2005) is also formally modeled as HaeberlenPastry.

- A correctness property of Pastry is identified and formulated as *CorrectDelivery* (Section 4.3.2), stating that there is at most one node responsible for delivering a key.

- The join protocol of Pastry is intensively analyzed and violations of *CorrectDelivery* in CastroPastry and HaeberlenPastry are discovered and analyzed (Sections 3.2 and 5.2).

- HaeberlenPastry, IdealPastry and LuPastry assign different statuses ("dead", "waiting", "ok" and "ready") to an overlay node according to its stage during join and readiness for delivering a key (Section 3.3.2). Only "ready" nodes are supposed to have consistent key mapping with each other and are allowed to deliver a message.

- IdealPastry summarizes the improvements based on analysis of CastroPastry and HaeberlenPastry and it is verified against the property *CorrectDelivery* through inductive proof of invariants, under the assumption that no nodes leave the network and no concurrent joins occur within the coverage region of a "ready" node and its direct neighbors.

- A formal specification of Pastry algorithm LuPastry (introduced in pseudocode in Section 3.3.3) is designed based on IdealPastry. The improvement is that a "ready" node adds the joining node as soon as it receives the join request and does not accept any new join request until it gets the confirmation that the current joining node is "ready". LuPastry is proved to be correct w.r.t. *CorrectDelivery* under the assumption that no nodes leave the network. The relevant invariants are introduced in Section 6.5. The safety property *CorrectDelivery* cannot be proved under any more relaxed assumption, due to possible counterexamples caused by network separation when particular nodes simultaneously leave the network.

- The model checker TLC is intensively used on analyzing and validating the Pastry models with three, four or five nodes. Simplified versions are designed to overcome the state explosion problem. Together with analysis details, best practices and experiences of using TLC are provided in Section 5.

- The interactive theorem prover TLAPS is used to complete the formal proof of IdealPastry and LuPastry that they conform to the property *CorrectDelivery* all the time. The proof serves at the same time as evidence of correctness of the formal model w.r.t. the verified property *CorrectDelivery* as well as a real world example demonstrating the possibility of using TLAPS for a large scale proof consisting of more than 10,000 proof steps.

- The case study proving that the protocols IdealPastry and LuPastry conform to *CorrectDelivery* gives evidence of possibility and the way to manually conduct the proof of a complex transition system for its correctness property. Chapter 6 introduces this approach showing step by step to what extent the assumptions hold and how the proof is completed.

1.5 Structure of the Thesis

The thesis is structured as follows. Chapter 2 provides some necessary background on structured *P2P* systems and formal verification as preliminaries for understanding the rest.

In Chapter 3, Pastry is explained on the design level, from CastroPastry based on Castro et al. (2004) using pseudocode, to HaeberlenPastry inspired by Haeberlen et al. (2005) using flow charts, to IdealPastry as a first verified model and finally to LuPastry together with pseudocode and some introduction of its proof.

Chapter 4 represents the formal model of LuPastry together with the desired properties. Chapter 5 illustrates how model checker TLC is employed to analyze the correctness property, with selected counterexamples which are explained in detail. It also gives the validation results of model checking. Chapter 6 explains the theorem proving approach for analyzing HaeberlenPastry and LuPastry and the verification approach to IdealPastry and LuPastry.

In Chapter 7, other systems realizing *P2P* are introduced and other formal verification approaches on network protocols are compared with the work of this thesis. Chapter 8 summarizes the thesis with its contribution and gives an outline of possible further research.

2 Foundations

2.1 Peer-to-peer System

In recent decades, the concept peer-to-peer (*P2P*) and its supporting techniques and services on the Web have brought about an evolution in the usage of the Internet. Compared to *Client/Server* systems, a *P2P* system has a low barrier to deployment, large scalability of participating nodes, and resilience to faults and attacks. These properties lead to a wide application of *P2P* systems in current information and communication technologies.

Above all, *P2P* systems (Schollmeier (2001), Oram (2001)) are widely used for sharing resources. NAPSTER[1] was the first successful business to use a *P2P* system for sharing music, with a central web page maintaining the entries. In contrast to NAPSTER, GNUTELLA (Chawathe et al. (2003)) has no centralized components operated by any single entity. Subsequently, *P2P* services such as BITTORRENT (Cohen (2003)) gained the most popularity for allowing downloads of bulk data.

P2P techniques have also raised a huge challenge to traditional TV and movie industries with media streaming by PPTV[2] and PPSTREAM[3]. The Chinese company XUNLEI[4] has combined bulk data downloading with media streaming to provide a wide range of services to more than 80 million users. According to Commission (2012), EU-funded research is helping content providers stream video to millions of viewers simultaneously using a fraction of the bandwidth of traditional methods.

There has been debate over changes brought to the Internet by *P2P* systems. On the one hand, legal issues including copyright infringement and child exploitation cause many countries to restrict the use of *P2P* services. On the other hand, the *P2P* concept is supported with the argumentation that it can protect personal data, since users store their private data only on their local computer and share the data only in predefined circles, as implemented in DIASPORA[5] and then also adopted by Google+[6] (Boeing (2012)).

Besides the challenges from legal issues, *P2P* systems may offer new opportunities and at the same time face new challenges surveyed in (Fersi et al. (2013)), especially when combined with mobile networks and wireless sensor networks, due to their promising scalability and self-stabilization.

[1] `http://www.napster.de/`
[2] `http://www.pptv.com`
[3] `http://www.pps.tv`
[4] `http://www.xunlei.com`
[5] `http://diasporaproject.org/`
[6] `http://plus.google.com`

Other applications of *P2P* systems include Dynamo (DeCandia et al. (2007)), used internally by Amazon for many of its services and applications. Dynamo provides a storage substrate using distributed hash tables (DHT) which will be explained later.

At present, one of the most successful applications of *P2P* is Skype (Microsoft (2013)), one of the largest voice-over-IP service providers. According to Gillett (2012), Skype is privileged to "serve 250 million active users each month and support 115 billion minutes of person to person live communications in a quarter alone". Its super-node mechanism gained attention after an outage of the service on December 23, 2010, when millions of users could not make phone calls according to marketerterryb (2010). Different possible reasons for this disruption are discussed in York (2010) and Heichler and Newman (2010). However, it remains unclear what the real reason was for the unavailability or why the subsequent huge numbers of log-ins made the system unstable for such a long time. This outage together with the one in Arak (2007) motivated this thesis to formally analyze and verify the correctness of *P2P* network protocols, with a particular focus on *churn*, i.e. concurrent joining and departure of peer nodes.

2.1.1 History, Concept and Definition

The idea of using the *P2P* concept to share files and resources can be traced back to the emergence of the Internet in late 60s, since the goal of the original ARPANET was to integrate different kinds of existing networks in such a way as to allow every host to exchange information equally. In addition, peer-to-peer file sharing can also find its implementation in the still used Domain Name System (DNS), initially designed in 1982 to share the file *hosts.txt*.

With the development of the World Wide Web, which promotes intensive use of web browsers, HTTP protocol and *Client/Server* architecture, and the business models provided by ADSL and cable modems, users of the Internet have become more familiar with browsing and downloading than exchanging information with others on the network.

Since the beginning of the 21st century, peer-to-peer applications like Napster, BitTorrent, PPTV and Skype have become popular due to the heavy load on servers of streaming data for billions of users.

Schollmeier (2001) in the first international conference on peer-to-peer computing defined peer-to-peer as follows:

> *"A distributed network architecture may be called a Peer-to-Peer (P-to-P, P2P, ...) network, if the participants share a part of their own hardware resources (processing prower, storage capacity, network link capacity, printers, ...). These shared resources are necessary to provide the service and content offered by the network (e.g. file sharing or shared workspaces for collaboration). They are accessible by other peers directly, without passing through intermediary entities. The participants of such a network are thus resource (service and content) providers as well as resource (service and content) requestors (server-concept)."*

This definition can be interpreted as a comprehensive introduction to *P2P* systems in the early 21st century, with its most popular applications to file, hardware and service sharing.

In Oram (2001), a *P2P* system is defined as:

> *"A self-organizing system of equal, autonomous entities (peers)", which "aims for the shared usage of distributed resources in a networked environment avoiding central services."*

This definition gives the essence of *P2P* system: self-organization and decentralization, later refined in Steinmetz (2005).

2.1.2 Classification

P2P systems are always compared with *Client/Server* distributed systems due to their opposite focus and topologies. In a *Client/Server* architecture, a server stands in the middle and provides resources to all of the clients. Clients are not aware of each other.

In contrast to *Client/Server* system, *P2P* systems allow the peers to communicate with each other. However, according to the degree of centralization, they are further classified into two categories: *hybrid* systems and *pure P2P* systems.

In pure *P2P* systems, all peers are equal and exchange information and resources cooperatively rather than being coordinated. Pure *P2P* systems can again be further classified into unstructured and structured systems as shown in the ontology in Figure 2.1.

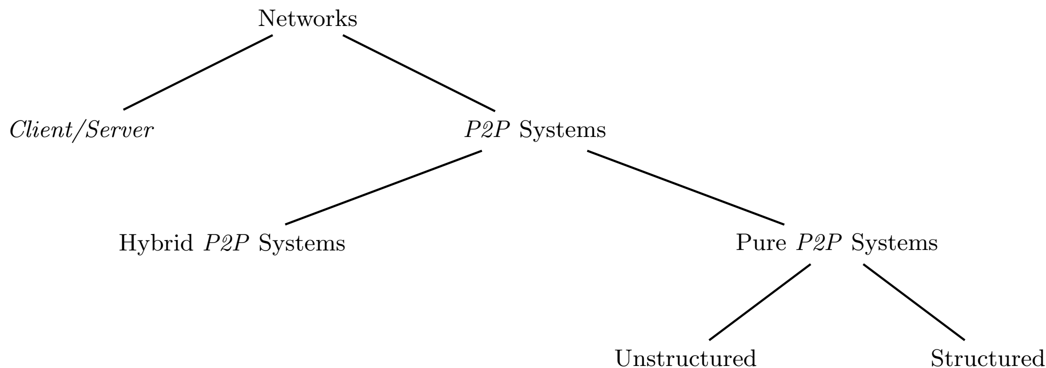

Figure 2.1: Ontology of networks.

In contrast to an unstructured *P2P* system, where nodes are connected in an adhoc way without any regulation, a structured *P2P* system normally relies on a virtual ring structure, where the identifiers of the nodes are ordered and nodes maintain their connections with their virtual neighbors according to their numerical identifiers. Hence "unstructured overlays are good at finding 'hay', while structured overlays are good at finding 'needles"', as pointed out by Rodrigues and Druschel (2010).

A hybrid *P2P* system is a compromise between *Client/Server* and a pure *P2P* system. It preserves the servers to some extent (some are called *Super Nodes*), so that they still stand in the middle of many peers to coordinate them, mainly helping them find each other, whereas the real communication and content exchange happens among the peers without intermediation from the server.

Since this thesis focuses on Pastry, a structured *P2P* system, the following sections give more details about this type. Comparison with other types will be summarized in Section 7.1. Other classifications with different perspectives and degrees of detail can be found in Milojicic et al. (2002), Androutsellis-Theotokis and Spinellis (2004) and Risson and Moors (2006).

2.1.3 Structured Decentralized System and DHT

Structured overlays provide efficient search (routing) functionality due to their structured identifier space and distributed data structures, which are also called *distributed hash tables* (DHTs), that provide data-centric routing, as discussed in Hellerstein (2003).

DHTs are designed to allow requests for a key to be correctly routed to the machine (referred to as a *node*) currently managing the value associated with the key, under the assumption that no global knowledge is available, nor permanent assignment of the mapping of keys to nodes.

Nodes in the *DHT* are assigned unique identifiers in a large numeric key space, say from 0 to $2^M - 1$, where 2^M is the capacity of the network. Such M can be chosen to be arbitrarily large in order to provide scalability. It is assumed that the nodes are assigned fairly and randomly enough (e.g. uniform distribution) with the identifiers from that space. The basic idea, as pointed out in Rodrigues and Druschel (2010), is to use a specific structure such that the identifier of a node determines its position within that structure and constrains its connections to other nodes.

Identifiers are used as addresses of nodes in the network as well as keys of the distributed content, which are divided by the participating nodes which take responsibility for the keys, such that the mapping from keys to nodes is a total functional mapping. This functional mapping is called a *placement function* in Rodrigues and Druschel (2010). The *DHT* has the same put/get interface as a conventional hash table, whereas the key/value pairs are distributed among the participating nodes in the structured overlay using a simple placement function. For example, a key can be mapped to the node whose identifier is the key's closest counter-clockwise successor in the key space. In this case the key space is interpreted to be a ring (i.e. the identifier zero succeeds the highest identifier from the space) to avoid the case that a key might be greater than all node addresses.

Various routing algorithms realize *DHT*, such as CAN by Ratnasamy et al. (2001), Tapestry by Zhao et al. (2004), Chord by Stoica et al. (2001) and Pastry by Rowstron and Druschel (2001). The algorithms have been introduced and compared in Androutsellis-Theotokis and Spinellis (2004) and Risson and Moors (2006). According to Hellerstein (2003), they all have the following properties:

- Small numbers of neighbors (low degree)

  Each node maintains information about the number of its neighbors. Typically this number is $M = log(2^M)$, where 2^M is the capacity of the key space as mentioned before, meaning that up to M update messages need to be sent upon arrival and departure of any node among the M nodes previously maintained by the node.

- Converging routes

  Each node should be able to find the appropriate next hop autonomously based on the locally maintained information (e.g. the neighbors) without help from other nodes. Moreover, the next hop should be always closer to the key than the current node.

- Minimal route hops (low diameter)

  The number of routing steps is not linear but logarithmic in the number of nodes on the ring, typically $M = log(2^M)$. For example, a node n receives the route request for a given key k and produces a route via a sequence of nodes whose identifiers share increasingly longer prefixes with the key. When the request lands at a node r, determined to be responsible for the target key according to its knowledge of neighboring nodes, then the request is answered by node r directly to node n. An example of such a routing algorithm, Pastry Castro et al. (2004), will be illustrated in detail in Figure 3.2 in Section 3.1.

- Resilience to churn (robustness)

  Churn refers to frequent arrival and departure of nodes. In the context of *DHT*, resilience against churn is crucial for making robust and reliable *P2P* systems. From the static point of view (without consideration of churns), all the *DHTs* with logarithmic degree (number of neighbors maintained by each node) and diameter (number of hops to find a key) w.r.t. total identifier space have comparable "dependability". In other words, they have more or less on the same level of robustness of routing according to Loguinov et al. (2003) and Gummadi et al. (2003). From the dynamic perspective, *DHT* dependability is sensitive to the underlying overlay maintenance algorithms. Risson and Moors (2006) compares different ways of handling *DHT* primitives of nodes (join, departure, maintenance, gossip, route etc.) and categorizes them based on the comparison. A conclusion of Rhea et al. (2003) claims that existing *DHT* systems suffer from high churn rates, which is refuted by Castro et al. (2004) by showing that the implementation MSPastry provides consistent joining and maintains dependability by continuous detection and repair.

Although *DHTs* facilitate finding the needle in a structured network, there are also drawbacks to these systems. First of all, *DHT* can only use key-based routing: no advanced search criteria can be used, such as the conjunction of keywords used in some unstructured *P2P* system. Besides, maintaining a correct *DHT* in a high-churn environment is expensive due to network traffic and local updates. In some applications,

finding the needles may not be worth the cost. For this reason, large data values are typically not inserted directly into a *DHT* but linked by indirect pointers which are inserted as the value corresponding to the hash key. In addition, many systems as mentioned before use precise key-based routing on a *DHT* layer only for coordination, and the dissemination of content is handled by other layers. Therefore, it would make sense to implement Pastry on the super nodes of a hybrid system such as SKYPE.

2.2 Formal Verification

Together with the growth of hardware, software and Internet technologies in their scale, functionality and ubiquitous application, the likelihood of subtle errors increases inevitably. Moreover, errors in critical systems may cost catastrophic loss of money, time or even human lives, as pointed out in Clarke and Wing (1996). Web pages Neumann (2013) and Dershowitz (2013) summarize those horror stories caused by software. Although IT projects fail due to combinations of various reasons, “badly defined system requirements”, “unmanaged risks” and “sloppy development practice” are three of the most important listed in Charette (2005).

One way of avoiding such problems is to apply formal methods to specify and verify the design of the software, as described in Clarke and Wing (1996). Use of formal methods does not guarantee correctness for all possible implementations. However, they can greatly increase the understanding of a system by revealing inconsistencies, ambiguities, and omissions that are often undetected by people writing the system requirements and developing the system without regard for some sloppy risks. In fact, formal methods are applied in various advanced industrial sectors, in particular in cost intensive or life critical systems such as avionics, air traffic control and medical devices, as summarized in Hinchey and Bowen (1999).

Formal verification in the context of software systems is similar to the concept used in relation to hardware systems defined in Seger (1992). It can be understood as the act of proving or disproving the intended correctness property of an algorithm underlying a system with respect to a formal specification or an implementation, using formal methods of mathematics. In the following, the basic taxonomies and relevant techniques are discussed in detail.

2.2.1 Specification, Validation and Verification

The concepts specification, validation and verification (often referred to as SV&V) are widely used in hardware and software verification communities as defined in IEEE (2011), where executable code is verified against an abstract design specification.

Verification and validation can be distinguished by two questions as explained in Baier et al. (2008). Verification is asking if we have made “the thing right”, i.e. does the product conform to the desired requirements? Validation is asking if we are making “the right thing”, i.e. if the product is adapted to the user’s actual needs. As pointed out in Baier et al. (2008), validation amounts to judging “whether the formalized problem statement (model + properties) is an adequate description of the actual verification problem”.

The verification process consists of static (structural) and dynamic (behavioral) aspects. For example, for a software product one can inspect the source code (static) and run against specific test cases (dynamic). Validation is usually conducted only dynamically, i.e. the product is analyzed by putting it through typical and atypical use case scenarios.

The framework of the formal verification approach of this thesis is illustrated in Figure 2.2. Generally speaking, formal verification is a complementary approach to the design of a software system. Given a list of requirements the system is supposed to fulfill and the predefined environment that the system is supposed to operate in, a designer describes the basic components of a system and their internal and external behaviors. In Figure 2.2, the system components (static) and their behaviors (dynamic) are modeled as a formal specification, the environment as assumptions, and the requirements as properties. The verification approach takes the formal model and assumptions as a starting point and verifies that the properties hold for the model under the assumptions. Modeling and validation as illustrated in Figure 2.2 are treated as complementary processes. Validation takes a formal model as input and checks if it models the informal description in terms of (its) intended functionality and properties.

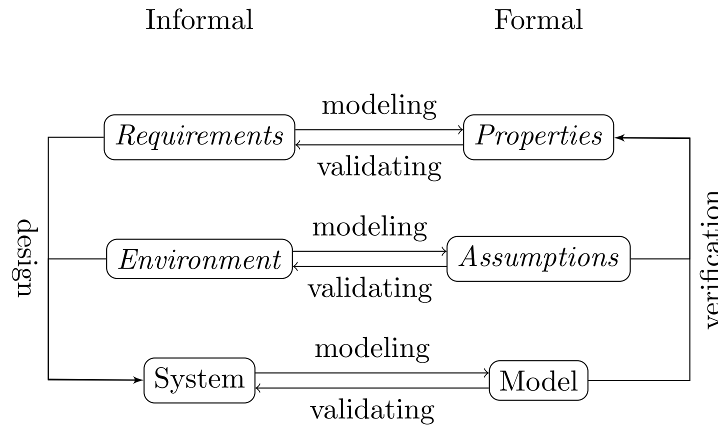

Figure 2.2: Framework of formal verification.

In the context of this thesis where formal verification is applied on a *P2P* system, these concepts (SV&V) play the same role. As illustrated in Figure 2.2, (formal) *model* (or formal specification) refers to the formal description of the system including the static model describing the data structures and the dynamic model describing the behaviors. Formal verification starts with modeling. According to Clarke and Wing (1996), modeling (or specification) is the process of “describing a system and its desired properties using a language with a mathematically defined syntax and semantics”. The formal specification of a *P2P* system is given in Chapter 4, together with its properties describing the desired requirements of the system.

Formal validation in this thesis is conducted through model checking with constructed assertions of impossibility of desired functionalities of the system, such as lookup

and join services of a key-based routing algorithm. By finding violations of these assertions, the model checker shows that the formal model is able to "do the right thing".

Formal verification in this thesis is carried out through both model checking analysis and a theorem proving approach. The goal of model checking is to find possible evaluations of the formal specification that violate the properties. Section 5.2 demonstrates some counterexamples found through model checking. The theorem proving approach aims to prove that the specification of the system does conform to the properties. Chapter 6 describes the theorem proving approach to verifying a *P2P* system against its safety properties. In contrast to model checking, a formal verification through theorem proving is more rigorous and precise and usually covers all possible evaluations rather than restricted instances. In the context of distributed systems, in particular *P2P* systems where an infinite transition system is used as a formal model, a theorem proving approach is necessary because model checking is no longer feasible due to state explosion, and is also challenging because the nature of message passing and distribution of data makes the construction and proof of global invariants subtle. For this reason, the following sections give some necessary background knowledge for better understanding the model checking and theorem proving approaches used in this thesis.

2.2.2 Model Checking and Theorem Proving

Model checking and theorem proving are the two predominant methods of verification, which "go one step beyond specification" (Clarke and Wing (1996)); they are used to analyze a system for desired properties. In the following, their principles and selective state-of-the-art approaches are introduced and a combined framework using TLA$^+$ is illustrated at the end as the methodology developed in this thesis to verify the *P2P* system at hand.

Model Checking

According to Clarke and Wing (1996), model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model via an exhaustive state space search that is guaranteed to terminate since the model is finite. In the case of verification of distributed systems, the model can become infinite but model checking techniques still find their application.

Relatively straightforward definitions of model checking are given in Clarke (2008) and the book Grumberg and Veith (2008):

> *" Let M be a Kripke structure (i.e. state-transition graph). Let f be a formula of temporal logic (i.e. the specification). Find all states s of M such that $M, s \models f$. "*

According to Baier et al. (2008), a typical model-checking process consists of three phrases: modeling, running and analysis, as illustrated in Figure 2.3.

In the *modeling* process illustrated in Figure 2.3, informal descriptions of the requirements, environment and system are coded in a formal specification language suitable for

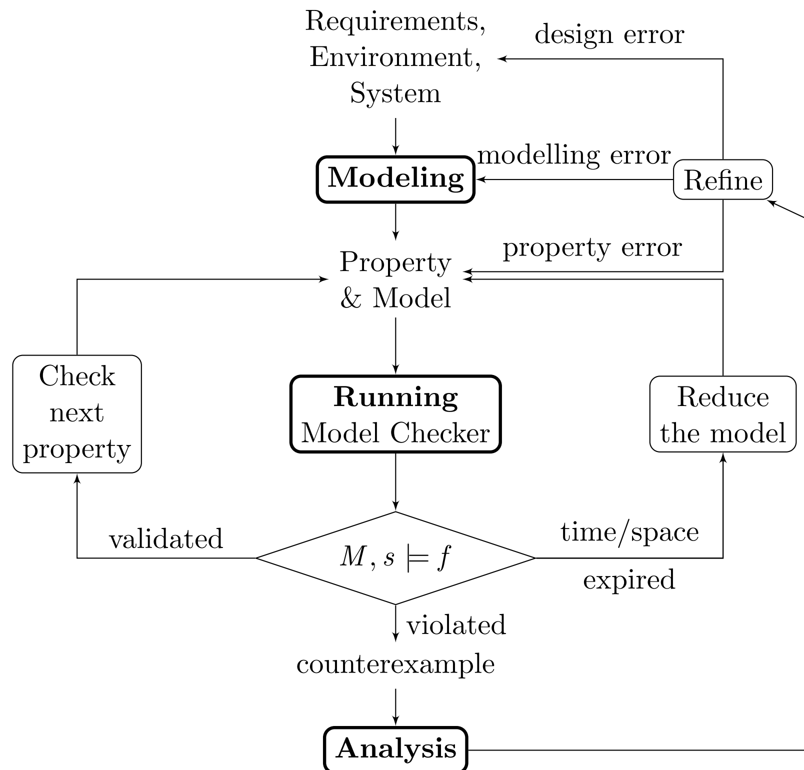

Figure 2.3: The model checking process.

mechanical model checking. The given system is typically modeled as a state-transition system (or automaton) consisting of a set of states and a set of transitions. States comprise information about the current values of variables. Transitions describe how the system evolves from one state to another.

Properties are typically modeled using *temporal logic*, a form of modal logic extending the traditional propositional logic with operators that refer to the behavior of systems over time. As discussed in Lamport (1980), properties are typically modeled by two kinds of temporal logic: linear-time logic (LTL) and branching-time logic (such as computation tree logic, CTL, later generalized by Emerson and Halpern (1986) as CTL*). Using either one of them, one can easily express the functional correctness properties (i.e. "does the system ever / always do what it is supposed to do?"), reachability (i.e. "is it possible to end up in a deadlock state?"), safety properties (i.e. "nothing bad ever happens"), and liveness (i.e. "something good will eventually happen"). Using LTL one can further express the fairness property (i.e. "does an event occur repeatedly under certain conditions?"). Using CTL one can further express the possibility of a certain event occurring sometime in the future along some computation path. CTL* is a logic that encompasses both CTL and LTL.

Running a model checker as illustrated in Figure 2.3 typically takes the state transition system M and a temporal formula f and determines whether the formula holds for all states s (either reachable states in the case of CTL or all sequences of executions in the case of LTL). If the formula is satisfied in every state by termination of the model checking, then the next property can be checked. When all properties have been checked the model is concluded to possess all desired properties.

As illustrated in Figure 2.3, on violation, the model checker typically provides a counterexample which can then be analyzed and reproduced by simulation. The analysis often leads to refinement of the model because the reason for this violation can be *modeling error*, *design error* or *property error* according to Baier et al. (2008).

A *modeling error* may occur when the formal model does not really reflect the design. Discovering a modeling error often improves our understanding of the design. Since the model had to be refined, the verified properties on the erroneous model may no longer be valid and the verification needs to be repeated.

Design error means that the counterexample corresponds to an intended run of the algorithm (or system) but violates a desired property. In such case, the design needs to be improved. The remodeled design will start a new modeling process, while the previous model and properties can be partially preserved.

Sometimes a *property error* occurs when the formal property does not reflect the informal requirement. In such case, only the corresponding property needs to be reformulated. As a result of the model checking analysis presented in Chapter 5.2, only design errors are reported and the other two types of error have already been corrected on the way to the final model. During model checking one typically analyzes finite instances of algorithms or systems, so even a successful run of the model checker does not mean that the system is correct under all circumstances, but only in that particular instance.

If the model checking process exceeds the threshold of running time or memory, usually due to *state-space explosion*, then the model checker is typically interrupted and

the model is supposed to be reduced. A typical technique is *Partial Order Reduction*. Partial order reduction is incorporated in different approaches, such as the *stubborn sets* of Valmari (1991), *persistent sets* of Godefroid (1991), or *ample sets* of Peled (1994). They differ in their actual details, but exploit *independent* events that are executed concurrently. The idea is to execute only one sequence of independent events because any other permutation would lead to the same overall result state. Two events are independent if all permutations of their executions result in the same global state. Further techniques can also be applied, such as abstraction in Clarke et al. (1994) or symmetry reduction in Clarke et al. (1993), among others. Another approach to model checking is the *Symbolic Model Checking*. Symbolic model checking, as shown in McMillan (1993), is based on *ordered binary decision diagrams* (OBDDs), which provide a canonical form for Boolean formulas that is often substantially more compact than conjunctive or disjunctive normal forms, and efficient algorithms have been developed to process them. In order to overcome the exponential growth of BDDs, Biere et al. (1999) introduces *bounded* model checking, which uses a propositional SAT solver in place of BDD manipulation techniques. As surveyed in Biere et al. (2003), bounded model checking is widely perceived as a complementary technique to BDD-based model checking.

Grumberg and Veith (2008) collects the dominant developments in model checking techniques through the 25 years till 2008. As pointed out in Baier et al. (2008), "model checking is an effective technique to expose potential design errors". The great advantage of model checking is that the verification process is fully automatic and the counterexamples are very useful for debugging purposes. Therefore, it is easy to understand, easy to integrate and hence widely adopted in industry. However it also has many limitations. Its primary disadvantage is that it does not in general scale to large systems due to its *state-space explosion* problem in state-exploration approach and it can never check *generalizations* (i.e. systems with an arbitrary number of components, or infinite-state systems) or reason about abstract data types. These drawbacks can be compensated by theorem proving, which will be introduced in the next section.

Theorem Proving

As introduced in Clarke and Wing (1996), theorem proving is a technique by which both the system and its desired properties are expressed as formulas in some mathematical logic given by a formal system defining a set of axioms and a set of inference rules.

Theorem proving has already seen several decades of development alongside different logics and their tools. Approaches vary not only in their choice among different classes of logics, such as classical logics (including propositional logic, first-order logic and higher-order logic) and modal logics (e.g. temporal logic, description logic, etc.) but also in the homogeneous reasoning techniques applied on them. Only machine-assisted (or *automated*) theorem proving approaches will be discussed in this thesis. According to the degree of automation, automated theorem proving approaches can be further classified as "interactive" or "fully automated". Since the verification approach of this thesis is based on first-order logic and set theory, only the relevant preliminaries are introduced in this section and some introduction of temporal logic will be given together with TLA$^+$

in the next. Further details of theorem proving approaches together with their logical foundations can be found in Fitting (1996) and Ben-Ari (2012).

The principle of applying methods of formal logic is to use unambiguous and precise mathematical description to model an ideal world where truth is unqualified and "there are no shades of grey", as mentioned in Fitting (1996). In the logical model, reasoning is carried out with *sentences* that are built up from primitive assertions about the world using connectives like *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$), *implies* ($\Rightarrow$), *every* ($\forall$), *some* ($\exists$), *equals* ($=$) and so on to connect and construct the primitive assertions. Sentences are *valid* if all interpretations are true. They are *provable* when "by logic alone" (i.e. purely syntactically, without evaluation of any possible interpretation as semantics) they can be derived from axioms by applying basic inference rules on this particular logic. The derivation (proof) steps are also called *logical consequences* according to Fitting (1996). The task of theorem proving is to find a proof (a sequence of logical consequences) of a property, stated as a sentence g using the axioms of the system (the predefined set of valid sentences S), rules, and possibly derived definitions and intermediate lemmas as arguments.

In a fully automated theorem proving approach, a proof procedure (Automated Theorem Prover, ATP) typically *proves* a sentence if it finds the proof by mechanical manipulation of the formulas. Depending on the logic applied, a proof procedure is called *sound* by Fitting (1996) if it proves only valid sentences (i.e. if a proof is found for sentence g, then g is valid) and *complete* if it proves every valid sentence (if the sentence g is valid, the prover will find a proof).

Many of the state-of-the art first-order automated theorem provers are based on superposition, a powerful extension of resolution techniques for first-order-logic with heuristics such as ordering and selection. Superposition is explained in Bachmair et al. (1992) and one of the most successful implementations is SPASS Weidenbach et al. (2009), introduced in Weidenbach et al. (1996). For the time being, many first-order ATPs such as Weidenbach et al. (2009), Schulz (2002), Riazanov and Voronkov (2001) and Prevosto and Waldmann (2006) are based on this calculus.

Recently, the Satisfiability Modulo Theory (SMT) approach described in Nieuwenhuis et al. (2006) successfully combined first-order reasoning with decision procedures for theories such as equality, integer and real arithmetic, arrays and bit-vectors. State-of-the-art systems include Yices by Dutertre and De Moura (2006), CVC3 by Barrett and Tinelli (2007) and Z3 by De Moura and Bjørner (2008).

The framework for using fully automated system provers for verification is in fact more or less the same as model checking approach illustrated in Figure 2.3 with the model checker itself replaced by an ATP. The process of interactive theorem proving is then slightly different because it allows human guidance in the search for the proof. Figure 2.4 illustrates the framework employed by an interactive theorem prover to construct a mechanically verified hierarchical proof. For each proof goal, an automatic step and a human-interacting step are conducted consecutively.

An interactive proof process starts with manual construction of the proof, which involves describing the proof goals g and providing their corresponding lemmas S that are either proved or hypothetically assumed. Given the proof goals g and a set of lemmas

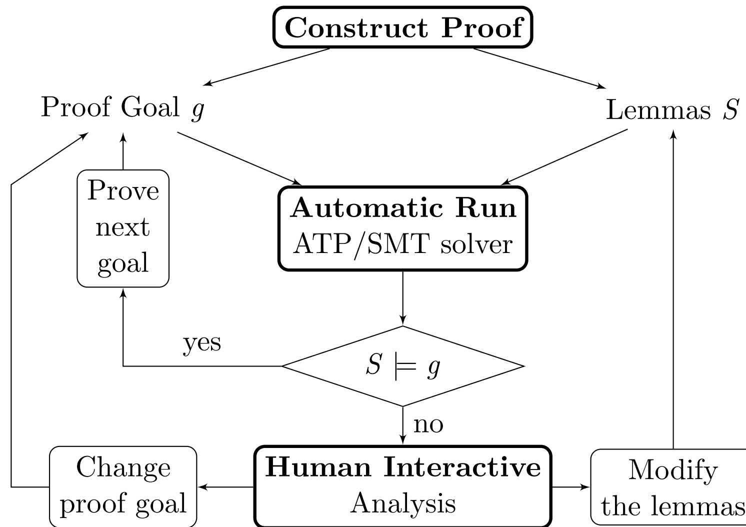

Figure 2.4: The interactive theorem proving process.

S as prerequisites written in a compatible formal specification language, an ATP and/or SMT solver can be launched as shown in Figure 2.4 to automatically deduce the proof using the embedded rules or calculi implemented in these provers. Here the lemmas can include axioms as well as derived formulas that can be applied to prove the current local proof obligation (goal).

If the proof goal g is mechanically derivable from the set of lemmas S as illustrated in Figure 2.4, the prover will reply with a proof verifying it. Then the next proof goal can be processed.

On the one hand, first-order-logic is in general semi-decidable; on the other hand, the theorems involved in the model make the problem more complex. Therefore, a threshold proving time is set on launching the prover. In the event that no proof is found within this limit, the input proof obligation, proof structure and supporting arguments (lemmas) need to be analyzed.

Sometimes it is necessary to *modify the lemmas*, i.e. to add more lemmas or to make some of the lemmas stronger or weaker. Afterward, the process shown in Figure 2.4 is repeated.

Sometimes the search space is simply so large that it is hard for the theorem prover to find the proof even if all needed lemmas are given. In this case, the proof goal will be changed by breaking it down into sub-proof steps using different proof structures such as case analysis, induction, backward reasoning or proof by contradiction. Then the sub-proof steps are verified recursively using the same loop in Figure 2.4. Sometimes the proof goal is invalid, normally discovered when the proof obligation has a very simple structure and it is easy to construct a counter example manually. In this case, the proof goal needs to be changed and the next proof sketch level needs to be modified, i.e. this invalid proof goal cannot be used as a lemma for higher level proofs. New proof sketches

are designed and corresponding sub-proof goals and lemmas are again to be verified using the same loop in Figure 2.4. The last two cases are summarized in Figure 2.4 by *change the proof goal* step.

Most interactive theorem proving (ITP) systems are based on higher-order-logic (HOL). State-of-the-art ITPs based on HOL include ISABELLE, introduced in Paulson (1989) and Nipkow et al. (2002), as well as *Coq*, introduced in Bertot and Castéran (2004).

Other ITPs based on first-order-logic with set theory include TLAPS, the proof system of TLA$^+$, which will be introduced in more details in the next section.

2.3 TLA$^+$ Specification Language

TLA$^+$, by Lamport (2002), is a formal specification language based on untyped *Zermelo-Fraenkel* (ZF) set theory with choice for specifying data structures, and on the Temporal Logic of Actions (TLA) for describing dynamic system behavior. Chapter 4 demonstrates the usage of TLA$^+$ as a formal specification language to specify a distributed algorithm. A summary of all language constructs of TLA$^+$ can be found in Lamport (2000). This section introduces only those parts of the TLA$^+$ language needed to understand this thesis. For more background on TLA$^+$, it is recommended to read Lamport (2002), Merz (2008) and Cousineau et al. (2012).

2.3.1 Constant Operators

Logic

First of all, TLA$^+$ inherits the syntax and semantics of standard first-order logic. Table 2.1 summarizes the syntax and the meaning. Here all the occurrences of x are bound in the expression p. TLA$^+$ requires that an identifier be declared or defined before it is used, and that it cannot be reused, even as a bound variable, in its scope of validity.

BOOLEAN	$\{$TRUE, FALSE$\}$
$\wedge, \vee, \neg, \Rightarrow$	and, or, not, implies
$\forall x : p$	for all interpretation of x, formula p holds
$\exists x : p$	there exists an interpretation of x, s.t. formula p holds
$\forall x \in S : p$	for all element x of set S, formula p holds
$\exists x \in S : p$	there exists an element x of set S, s.t. formula p holds

Table 2.1: Syntax of First-Order Logic in TLA$^+$

Sets

Elementary set theory is based on a signature that consists of a single binary predicate symbol $\in$ and no function symbols. Nevertheless, TLA$^+$ provides some syntactic sugar for set expressions, operations and comparisons as summarized in Table 2.2.

$=, \neq, \subseteq$	equal, not equal, subset or equal
$\in, \notin$	a member of, not a member of
$\cup, \cap, \setminus$	union, intersection, set difference
$\{e_1, \ldots, e_n\}$	Set consisting of elements e_i
$\{x \in S : p\}$	Set of elements x in S satisfying p
$\{e : x \in S\}$	Set of elements e such that x in S
SUBSET S	Set of subsets (the power set) of S

Table 2.2: Syntax of Set Theory in TLA$^+$

For a finite set S, TLA$^+$ provides a standard module introducing the expression *Cardinality*(S) to denote the number of elements in S. The expression *Cardinality*(S) is abbreviated to $|S|$ in this thesis.

Naturals and Arithmetic

There are several standard modules written in TLA$^+$ to provide the primitives of common modeling tasks. The standard module *Naturals* of the TLA$^+$ library provides operators representing natural numbers and arithmetic operators.

Table 2.3 summarizes the symbols and definitions in these modules. Here the variables a and b are natural numbers.

$a + b,\ a - b,\ a * b,\ a^b$	binary operators plus, minus, multiplication and exponents
$a < b,\ a \leq b,\ a > b,\ a \geq b$	binary comparisons
$a..b$	$\{n \in \mathbb{N} : a \leq n \leq b\}$
$a\%b$	a mod b, such that $0 \leq a\%b < b$
$a \div b$	division, such that $a = b * (a \div b) + (a\%b)$

Table 2.3: Syntax of Arithmetic in TLA$^+$

Since the symbol $\div$ is defined for *Naturals* and the symbol / (divisions) is defined for *Reals* in TLA$^+$, this thesis uses only $\div$ for division as defined in Table 2.3.

The Choose Operator

TLA$^+$ heavily relies on Hilbert's choice operator. An expression CHOOSE $x : p$ means taking some arbitrary value x that makes the expression p TRUE, or in other words, satisfies the property p. If no such value exists, CHOOSE will simply pick an arbitrary unspecified value. In practice, reasoning about expressions involving CHOOSE requires proving that there exists some x satisfying p. Note that all occurrences of x in the term p are bound. As syntactic sugar, the TLA$^+$ expression CHOOSE $x \in S : p$ means CHOOSE $x : (x \in S) \wedge p$, which means take value x, which is at the same time an element of S and satisfies the property p, if such a value exists.

Functions

Besides set and logic operations, functions are a convenient way to represent different kinds of data structures. In contrast to certain traditional constructions of functions within set theory, which construct functions as special kinds of ordered pairs, TLA$^+$ assumes functions to be primitive and assumes tuples to be a particular kind of function.

The application of function f to an expression e is written as $f[e]$. The set of functions whose domain equals X and whose co-domain is a subset of Y is written as $[X \rightarrow Y]$. The expression $x \in \text{DOMAIN } f$ denotes that the element x is an instance of the domain of the function f. The expression $[x \in X \mapsto e]$ denotes the function with domain X that maps any $x \in X$ to e; again, the variable x must not occur in X and is bounded to e. In fact, this expression can be understood as the TLA$^+$ syntax for a lambda expression $\lambda x \in X : e$.

TLA$^+$ introduces the notation: $[f \text{ EXCEPT } ![e_1] = e_2]$. It means that the resulting function, say f_{new}, is equal to the function f except at the point e_1, where its value is replaced with e_2, namely $f_{new}[e_1] = e_2$. Here both e_1 and e_2 are expressions.

Tuples

A tuple is a function with domain $1..n$, for some natural number n. Hence, the expression $\langle e_1, \ldots, e_n \rangle$ denotes the n-tuple whose i^{th} component is e_i. The function application $e[i]$ here denotes the ith component of tuple e. The cross product of n sets $S_1 \times \cdots \times S_n$ represents the set of all tuples with i^{th} component in S_i.

String and Records

TLA$^+$ provides records to facilitate access to components of tuples by giving them "names" as strings. In fact, strings are defined as tuples of characters and records are represented as functions whose domains are finite sets of strings. Since tuples, strings and records are all based on functions, the update operation on functions can be applied to them as well. Furthermore, TLA$^+$ offers more abbreviations for record operations. $e.h$ denotes the h-component of record e. $[h_1 \mapsto e_i, \ldots, h_n \mapsto e_n]$ denotes the record whose h_i component is e_i. $[h_1 \in S_1, \ldots, h_n \in S_n]$ denotes the set of records with field names h_i and whose respective values are in S_i.

2.3.2 Modules

For modeling large systems, specifications in TLA$^+$ can be organized in separate *modules*. A *module* is a basic unit of a TLA$^+$ specification, which typically contains *declarations*, *definitions* and *assertions*.

In TLA$^+$, an identifier must be declared or defined before it is used, and it cannot be reused, even as a bound variable, in its scope of validity. Every symbol must either be a built-in operator of TLA$^+$ (like $\in$) or it must be declared or defined. The scope of its validity is normally within the module, say M_1, where it is defined, but it can be extended to other modules, say M_2, by declaration of that module using statements of

the form EXTENDS M_1 in the module M_2. A module could also be an instantiation of another module, but this feature is not used in this thesis. Form and details can be found in Lamport (2002).

Declarations

A module may extend other modules, importing all their declarations and definitions, constant parameters and variables. Constant parameters represent entities whose values are fixed during system execution, although they are not defined in the module because they may change from one system instance to the next. Note that there are no type declarations because TLA$^+$ is based on set theory and all values are sets.

Variable parameters represent entities whose values may change during system execution. They correspond to program variables in this sense.

A specification of a distributed system can be modularized to static and dynamic modules. The static modules declare the constant parameters as primitive data structures. Then these static modules are extended to the dynamic modules where variable parameters are declared.

Definitions

In TLA$^+$, definitions are given in the form:

$$Op(arg_1, \ldots, arg_n) \triangleq exp$$

Here, Op is defined to be the operator such that $Op(e_1, \ldots, e_n)$ equals exp, where each arg_i is replaced by e_i. In case no arguments are given, i.e. $n = 0$, it is written as $Op \triangleq exp$. This shows the essence of a definition: it assigns an abbreviation (syntactic sugar, synonym) to an expression, which will never change its meaning (semantics) in any context. For example, $x \triangleq a + b$ means that x is syntactic sugar for "sum of a and b". Therefore, the expression $x * c$ is equal to $(a + b) * c$, instead of $a + b * c$ $(= a + (b * c))$.

For modeling distributed systems, complex data structures and their operators are typically introduced using definitions without or with arguments respectively. Note that argument arg_i is declared locally within the definition and its scope is only within exp.

Although definitions are used to define system operations in a way which appears similar to common programming languages, these operations have no "side effects", they are all functional. In TLA$^+$, a symbol or an identifier can only be declared or defined once. From then on, it will never change its meaning. Therefore, TLA$^+$ language is more assertional than operational. From the programming language perspective, it is declarative but not imperative.

Assertions

In TLA$^+$, assertions include assumptions, lemmas and theorems. For modeling distributed systems, assumptions can be either explicitly stated or implicitly encoded as

preconditions of the actions. The explicit way helps the model checker to constrain the number of explorable states. The implicit way helps to give a more plausible implementation of the real system. Therefore, the thesis tries to avoid using explicit assertions for assumptions.

Lemmas and theorems typically assert causal relationships between definitions. There is no formal distinction in TLA$^+$ between a system specification and a property: both are expressed as formulas of temporal logic. Hence, asserting that specification S has property F amounts to claiming validity of the implication $S \Rightarrow F$. In this thesis, lemmas and theorems are typically written in this form. The proof language of TLA$^+$ also introduces the notion of *sequent* for stating assertions, which will be introduced later.

2.3.3 Other TLA$^+$ Syntax

TLA$^+$ uses syntactic sugar to extend the primitive functionality, which facilitates the readability, in particular for software engineers who are used to conventional programming language.

$$\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2$$

The expression is equivalent to $(p \wedge e_1) \vee (\neg p \wedge e_2)$

$$\text{LET } d_1 \triangleq e_1 \ldots d_n \triangleq e_n \text{ IN } e$$

The expression LET defines local operators, while IN encloses the region where these are used.

$$\begin{array}{lcl} \wedge\ p_1 & \text{conjunctions and disjunctions} & \vee\ p_1 \\ \wedge\ \vdots & & \vee\ \vdots \\ \wedge\ p_n & & \vee\ p_n \end{array}$$

Conjunctions and disjunctions in TLA$^+$ typically start with a conjunction symbol or disjunction symbol, which is in fact cruft in a conventional logical expression. But this kind of expression gives a better overview of the hierarchical structure of the logical formula. For example, instead of writing $(d_1 \vee (b_1 \wedge b_2)) \wedge c_1 \wedge c_2$, it is preferable in TLA$^+$ to use the following expression:

$$\begin{array}{l} \wedge \vee\ d_1 \\ \quad \vee \wedge\ b_1 \\ \quad\quad \wedge\ b_2 \\ \wedge\ c_1 \\ \wedge\ c_2 \end{array}$$

2.3.4 Transition System

A distributed system is typically specified as a transition system with initial state *Init* and its transitions, which are called *actions* in TLA$^+$.

Variables

The variables for modeling a distributed system are typically defined as functions whose domain are the processes (or nodes) of the system and values are the local instances of specific data structures. Recall that TLA$^+$ does not have types. Type invariants are typically introduced with proofs. They normally have the following form:

$$\begin{aligned} \mathit{TypeCorrectness} \triangleq\ & \land\ \mathit{var}_1 \in [\mathit{Node} \rightarrow S_1] \\ & \land\ \ldots \\ & \land\ \mathit{var}_n \in [\mathit{Node} \rightarrow S_n] \end{aligned}$$

Here var_i is the variable name such that $\mathit{vars} = \langle \mathit{var}_1, \ldots, \mathit{var}_n \rangle$. Node is typically the data structure of distributed entities in the system. S_i is the set of particular data structures.

Different Levels of Formulas

In TLA$^+$, formulas have different levels. A *constant* formula describes the properties of constants and constant parameters but they do not contain state variables. For example, $I \in \textsc{subset}\ \mathit{Nat}$ is a constant formula assuming that I does not contain state variables, which expresses that the data structure I is a subset of the natural numbers.

A *state* formula describes the properties of state variables. For example, $x \in I$ (where x is a state variable) is a state formula expressing that this variable is an instance of the data structure I.

An *action* formula describes the changes of state variables after a transition. For example, $x' = x + 1$ is an action formula expressing that the state variable x is incremented by 1 after this transition.

In an action formula, the primed variable refers to the variable after change and the unprimed variable refers to its original value.

A *temporal* formula describes the properties of state variables throughout the sequence of executions of transitional actions. For example, $\Box(x \in I)$ forms a temporal formula out of a state formula, which states that x is always a member of I. $\Box[A]_e$ forms a temporal formula out of an action formula A, specifying that every transition either satisfies the action formula A or leaves the expression e unchanged. The index e is an expression stating the set of stuttering variables (whose values are not changed by transition) if no actions are executed. Typically the e is a sequence of variables vars such that $\mathit{vars} = \langle \mathit{var}_1, \ldots, \mathit{var}_n \rangle$.

Initial States

The formula *Init* consists of the initial assignments of all variables, and typically takes the following form.

$$\begin{aligned} \mathit{Init} \triangleq\ & \land\ \mathit{var}_1 = [d \in \mathit{Node} \mapsto v_1] \\ & \land\ \ldots \\ & \land\ \mathit{var}_n = [d \in \mathit{Node} \mapsto v_n] \end{aligned}$$

Here the var_i is the variable name such that $vars = \langle var_1, \ldots, var_n \rangle$. The expression v_i in each assignment is the initial value for the particular variable.

Transition Rules as Actions

The formula *Next* defines the transition rule, which is typically a disjunction of all the actions. Each action formula is a first-order formula containing unprimed as well as primed occurrences of the state variables, which refer respectively to the values of these variables in the states before and after the action.

An action for modeling distributed systems in TLA$^+$ typically has the form:

$$\begin{aligned} ActionName(arg_1, \ldots, arg_m) \triangleq\ & \land\ precondition \\ & \land\ var_1' = [var_1 \text{ EXCEPT } ![arg_i] = v_1] \\ & \land\ \vdots \\ & \land\ var_k' = [var_n \text{ EXCEPT } ![arg_i] = v_k] \\ & \land\ \text{UNCHANGED } \langle var_{k+1}, \ldots, var_n \rangle \end{aligned}$$

An action may or may not have arguments which can be used only in the scope of its definition. An action typically consists of conjunctions of formulas which have different purposes. The formula *preconditions* can be a conjunction of several formulas, which serve as preconditions for the action, or so-called enabling conditions.

The formulas below the enabling conditions are the post-conditions of the actions, which define how state variables (var_i, i from 1 to k) of the transition system are altered using the update mechanism of function introduced before. Note that the value of the variable is a function mapping each element of the domain to a certain value. The mechanism changes the variables of only one element, arg_i, which is a parameter of the action definition. The expression v_i is the new functional value of that variable assigned to that particular element arg_i.

Suppose the total number of variables is n, then the rest of the variables, which are not modified by this action, should be summarized in UNCHANGED $\langle var_{k+1}, \ldots, var_n \rangle$, which is shorthand for the formula $\langle var_{k+1}, \ldots, var_n \rangle' = \langle var_{k+1}, \ldots, var_n \rangle$.

Note that making subsequent changes to a state variable in an action logically means that this variable is equal to different values, which is obviously a FALSE statement. A FALSE formula in an action which is a conjunction means that this action is in fact by definition FALSE and therefore will never be enabled. For this reason, modifications of one variable are always defined using one statement as shown in the formula above.

Temporal Formulas

A specification of the system can be written in the form:

$$Init \land \Box[Next]_{vars}$$

This form of system specification is sufficient for specifying the safety part of a system. The symbol $\Box$ is a temporal operator meaning that the formula following it is

"always" true, i.e. in all states of an infinite sequence of states. Here the overall formula means that all runs start with a state that satisfies the initial condition *Init*, and that every transition either does not change *vars* (defined as the tuple of all state variables: $vars \triangleq \langle var_1, \ldots, var_n \rangle$) or corresponds to a system transition as defined by *Next*. In general, the index f of a formula $[N]_f$ can be any state formula, not just a tuple of variables. This is important for refinement proofs but not used in this thesis.

2.3.5 Property, Inductive Invariant and Proof

In TLA$^+$, properties of a system (i.e. *Prop*) are also described as formulas. The statement that a system has a property is expressed as THEOREM $Spec \Rightarrow Prop$ and this asserts that the implication is valid. This assertion is normally assigned a name, in order to be referenced as a lemma in proof arguments. The names of such theorems, invariants and lemmas do not have "arguments" because they have implicitly the complete state variables as arguments. This thesis distinguishes *system properties* from *data structure properties*. Data structure properties are normally facts (lemmas or theorems) whose assertion is a constant formula, i.e. that does not contain state variables.

On the contrary, a system property describes the system behavior with respect to the state variables. It can be a *state property* expressed by a state formula, which describes particular relationships between different state variables in a state; it can be also a *temporal property* expressed by a temporal formula. Temporal properties can express safety or liveness property, but here in this thesis, only safety properties are considered.

A particular state formula

$$\text{ENABLED A}$$

for an action formula A is obtained by existential quantification over all primed state variables occurring in A. It is true at a state s whenever there exists some successor state t such that A can be executed at state s, which leads to state t, more precisely, such that the action formula A holds when all unprimed state variables are evaluated at state s and all primed state variables are evaluated at state t.

If a temporal operator $\Box$ is added in front of a state formula p, it is turned into a temporal formula. $\Box p$ means that the state formula p holds for all reachable states. In such cases, property p is also called an *invariant*.

Invariants characterize the set of reachable states during system execution. In order to prove $\Box p$, for a state formula p, the invariant p must be preserved by every possible system action. It is usually necessary to strengthen the assertion and find an "induction hypothesis" that makes the proof go through. This idea is embodied in the following invariant rule.

$$\frac{Init \Rightarrow Inv \qquad Inv \land [Next]_{vars} \Rightarrow Inv' \qquad Inv \Rightarrow p}{Init \land \Box[Next]_{vars} \Rightarrow \Box p} \text{ Invariant Rule}$$

Here *Init* stands for the initial state, *Next* is the disjunction of all actions. In order to prove that the invariant p holds in all reachable states of the system starting from the

initial state *Init*, it is sufficient to show that the initial state conforms to an inductive invariant *Inv* and any system step (that either corresponds to action *Next* or does not change the system variables *vars*) preserves *Inv*. The prime here stands for the primed variable as introduced in the action formula. Besides, the invariant *Inv* implies p. The subtle step is to find this inductive invariant *Inv*. Typically, inductive invariants contain interesting "design" information about the model and represent the overall correctness idea. This thesis will illustrate how inductive invariants can be discovered for a non-trivial algorithm.

2.3.6 Basics of the TLA$^+$ Proof Language

TLA$^+$ uses different notions to construct the proof. In the following, the basic notions are introduced via a simple proof of Cantor's theorem.

Given the definitions

$$\begin{aligned} \mathit{Range}(f) &\triangleq \{f[x] : x \in \text{DOMAIN } f\} \\ \mathit{Surj}(f, S) &\triangleq S \subseteq \mathit{Range}(f) \end{aligned}$$

of the range of a function and the notion of a function f being surjective for set S, Cantor's theorem can be stated as

$$\text{THEOREM } \mathit{Cantor} \triangleq \forall S : \neg\exists f \in [S \rightarrow \text{SUBSET } S] : \mathit{Surj}(f, \text{SUBSET } S)$$

Figure 2.5 shows a hierarchical proof of that theorem. Proof steps may have labels of the form $\langle d \rangle n$, where d is the depth of the step and n is a freely chosen name (numbers as in this example). The depth is also indicated by indentation in the presentation of the proof. A proof can be conducted in different ways. In TLA$^+$, users choose how to decompose a proof, starting a new level of proof that ends with a QED step establishing the assertion of the enclosing level. Assertions at the leaves of the proof tree are passed to the automatic provers of TLAPS. Here in this example it proves by contradiction.

TLA$^+$ uses *sequent* in the form of ASSUME ... PROVE ... to describe the proof. Step $\langle 1 \rangle 1$ reformulates Cantor's theorem as a sequent that assumes existence of a surjective function from S to SUBSET S, in order to prove a contradiction. Step $\langle 1 \rangle 2$ deduces the assertion of the theorem from that sequent.

The level-2 proof starts by picking some such surjective function f and then shows that it cannot be surjective. Hence $\langle 2 \rangle 3$ deduces a contradiction. The main argument of the proof is shown in the level-3 step, which defines the diagonal set $D \subseteq S$, which by definition cannot be in the range of f.

The keywords OBVIOUS, BY and DEF state that the proof step can be automatically verified by the back-end prover either directly, or by using the lemmas listed after BY and unfolding the definitions listed after DEF.

An alternative to ASSUME ... PROVE ... is to use SUFFICES ASSUME ... PROVE ..., which provides the *backwards chaining* proof tactic. In this example, using the latter expression will lift the proofs on the level of $\langle 2 \rangle$ to sequential proof steps after $\langle 1 \rangle 1$ in order to decrease the depth of the proof tree.

THEOREM $Cantor \triangleq \forall S : \neg \exists f \in [S \rightarrow \text{SUBSET } S] : Surj(f, \text{SUBSET } S)$
PROOF
$\langle 1 \rangle 1.$ ASSUME NEW S,
$\quad\quad \exists f \in [S \rightarrow \text{SUBSET } S] : Surj(f, \text{SUBSET } S)$
$\quad$ PROVE FALSE
$\quad \langle 2 \rangle.$PICK $f \in [S \rightarrow \text{SUBSET } S] : Surj(f, \text{SUBSET } S)$
$\quad\quad$ OBVIOUS
$\quad \langle 2 \rangle 2. \neg Surj(f, \text{SUBSET } S)$
$\quad\quad \langle 3 \rangle 1.$DEFINE $D \triangleq \{x \in S : x \notin f[x]\}$
$\quad\quad \langle 3 \rangle 2. D \in \text{SUBSET } S$
$\quad\quad\quad$ OBVIOUS
$\quad\quad \langle 3 \rangle 3. D \notin Range(f)$
$\quad\quad\quad$ BY DEF $Range$
$\quad\quad \langle 3 \rangle 4.$QED BY $\langle 3 \rangle 2, \langle 3 \rangle 3$ DEF $Surj$
$\quad \langle 2 \rangle 3.$QED BY $\langle 2 \rangle 2$
$\langle 1 \rangle 9.$QED BY $\langle 1 \rangle 1$

Figure 2.5: Proof of Cantor's Theorem in TLA$^+$.

A skeleton TLA$^+$ proof is shown in Section 6.6 to demonstrate how the constructs are used for proving invariants. More details about the TLA$^+$ proof language can be found in Lamport (2012b) and Cousineau et al. (2012).

3 Pastry Protocol

As introduced in Section 1.1, Pastry Rowstron and Druschel (2001) is an overlay network protocol implementing a scalable distributed hash table on a ring structure, where an active Pastry node is responsible to the keys numerically close to it. Since nodes can dynamically and concurrently join as well as leave the network without notification, it is not trivial to show the safety property *CorrectDelivery*, which states that there can never be two different Pastry nodes that consider themselves responsible for any single key.

This chapter illustrates the analysis of Pastry w.r.t. *CorrectDelivery* on an abstract level for a network protocol designer to understand the existing problems and the way to improve the network protocol through verification.

Section 3.1 explains the basic idea of Pastry, together with the algorithms in pseudocode. The pseudocode presents CastroPastry in an imperative way, which is an improved version of Pastry in Castro et al. (2004) with some necessary details added and bugs found during model validation fixed.

Section 3.2 focuses on analyzing the join protocol of Pastry, which is supposed to maintain a consistent local view of each node, and at the same time route and eventually deliver a message, despite concurrent joining of nodes. The improvement of lease granting protocol proposed by Haeberlen et al. (2005) is illustrated as flow charts and model checked as part of another version of Pastry, the HaeberlenPastry. Unfortunately, HaeberlenPastry cannot guarantee the safety property. More details of model checking the safety properties and beyond will be illustrated later in Chapter 5.

After that, Section 3.3 presents the final version of Pastry, the LuPastry, which solves all the problems discussed in Section 3.2, together with pseudocode in the same notation as CastroPastry in Section 3.1 for a better comparison. The detailed formal model of LuPastry will be illustrated in Chapter 4. LuPastry is formally verified against the safety property *CorrectDelivery*, under the assumptions that no nodes leave the network and an active node can help at most one node to join. Section 3.4 explains the motivation for these assumptions. More details of the formal proof are found in Chapter 6.

Finally, the contributions to the Pastry protocol are summarized in Section 3.5.

3.1 Algorithms of Pastry

3.1.1 Basic Idea of Pastry

Pastry (Rowstron and Druschel (2001); Castro et al. (2004); Haeberlen et al. (2005)) is an overlay network protocol implementing distributed hash tables (DHTs). A network

node in Pastry is assigned a logical identifier from an ID space (I) of natural numbers in the interval $[0, 2^M - 1]$ for some natural number M. The ID space constructs a virtual ring topology, where the neighbors of a node are those with the numerically closest Pastry identifiers. That is, $2^M - 1$ is the neighbor of 0. The IDs serve two purposes. First, they are the logical network addresses of nodes. Second, they are the keys of the hash table. In particular, an active node is responsible for delivering the keys that are numerically close to its network ID, i.e. it provides primary storage for the hash table entries associated with these keys. Key responsibility is divided equally according to the distance between two neighbor nodes. If a node is responsible for a key, it is said to *cover* the key.

The most important sub-protocols of Pastry are *join* and *lookup*. The join protocol eventually adds a new node with an unused network ID to the ring. The lookup protocol delivers the hash table entry for a given key. An important correctness property of Pastry is *CorrectDelivery*, requiring that there is always at most one (active) node responsible for a given key. Guaranteeing this property in the presence of spontaneous arrival and departure of nodes is non-trivial. Nodes may concurrently join the network or simply drop off, and Pastry is expected to be robust against such changes, i.e. *churn*. For this reason, every node holds two *leaf sets* of size L containing its closest neighbors to either side (L nodes to the left and L to the right). A node also holds the hash table content of its leaf sets neighbors. If a node detects, e.g. by a ping, that one of its direct neighbors dropped off, then the node takes actions to recover from this state. So the value of L is relevant for the amount of "drop off" and the fault tolerance of the protocol.

The Routing Algorithm

A lookup request must be routed to the node responsible for the key. Routing using the leaf sets of nodes is possible in principle, but results in a linear number of steps before the responsible node receives the message. Therefore, on top of the leaf sets of a node a routing table is implemented that enables routing in a number of steps logarithmic in the size of the ring.

The routing table contains $M \div B$ rows and 2^B columns, where M denotes the initial cardinality of the identifier space of Pastry. B is a configurable parameter, defining the base of the digits representing the identifier of a node. Node identifiers are thought of as a sequence of digits with base 2^B. The entry in row r and column c in the routing table is either empty (*null*) or a node identifier, which shares its first r digits with the node which this routing table belongs to, and whose $(r + 1)$th digit equals c. A more detailed explanation of the structure of the routing table and its influence on efficient routing can be found later in Section 4.1.3 or directly from Castro et al. (2004).

Pastry routes a message by forwarding it to nodes that match progressively longer prefixes with the destination key. The routing mechanism is demonstrated here with an example and will be explained in detail later in Section 3.1.3 with pseudocode.

Figure 3.1 illustrates the process of routing a message. Assume that there are 6 nodes in the ring: 1030, 1032, 1110, 2310, 3131 and 3200, located according to their numerical order. The identifier 3203 in italics represents a key. The length of each

side of the leaf sets is 1 ($L = 1$). All nodes are connected to their neighbors (solid bidirectional arrows). The dotted arrow represents the route step. The white sectors around 1032 and 3200 demonstrate the coverage ranges of these two nodes, which are illustrated explicitly for later reference. Together with the figure, Table 3.1 illustrates how the entries of the routing table of node 1032 should look, while Table 3.2 shows an example of the routing table of node 1032. In all the examples, the base is set to 4 ($B = 2$) and the cardinality of the identifier space is 2^{16}.

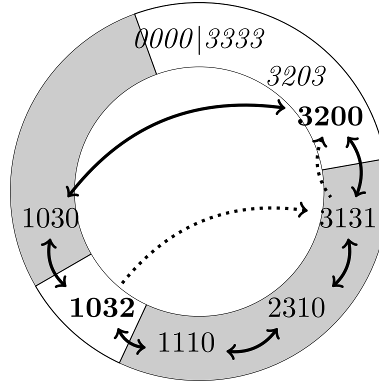

Figure 3.1: Pastry routing example.

In this example, node 1032 receives a lookup message for the key 3203. It first checks if it covers the key itself, so that it can answer the lookup request directly. Otherwise, it checks if it is within its coverage, such that a member of its leaf set could deliver it. As shown in Figure 3.1, the key 3203 lies outside the coverage of node 1032 and furthermore, it does not lie between the leftmost node and the rightmost node of its leaf sets, which are 1110 and 1030 respectively. The second step is to find the appropriate next hop from the routing table entries that share one more digit with the key than the node itself. Querying its routing table as shown in Table 3.2, node 1032 finds a more distant node 3131, which shares one more digit with the key than itself, and forwards the message to it. Node 3131 repeats the same steps and forwards the message to node 3200. Finally, the lookup message is answered by node 3200, which covers the key 3203. In this case, node 3200 *delivers* the lookup request for the key 3203.

The Join Protocol in CastroPastry

In Figure 3.2 and later flowcharts, node shapes are used in their standard sense: a rounded rectangle represents an action; a rhombus represents a decision; an ellipse represents a state, containing state variables with their values.

This flowchart presents the join protocol of CastroPastry, based on Rowstron and Druschel (2001) and Castro et al. (2004). An inactive node j announces its interest in joining the ring by performing the action Join(j, i). Action RouteJReq(i, j) will then route its join request to the closest active node k_0 just like a “Lookup” message, via intermediate nodes i, treating j as the key. Upon receiving a request in the form of

0*	1*	2*	3*
10*	11*	12*	13*
100*	101*	102*	103*
1030	1031	1032	1033

Table 3.1: Mask of routing table of node 1032 in base 4 (B=2)

null	1032	2310	3131
1032	1110	null	null
null	null	null	1032
null	null	1032	null

Table 3.2: Example of routing table of node 1032 in base 4 (B=2)

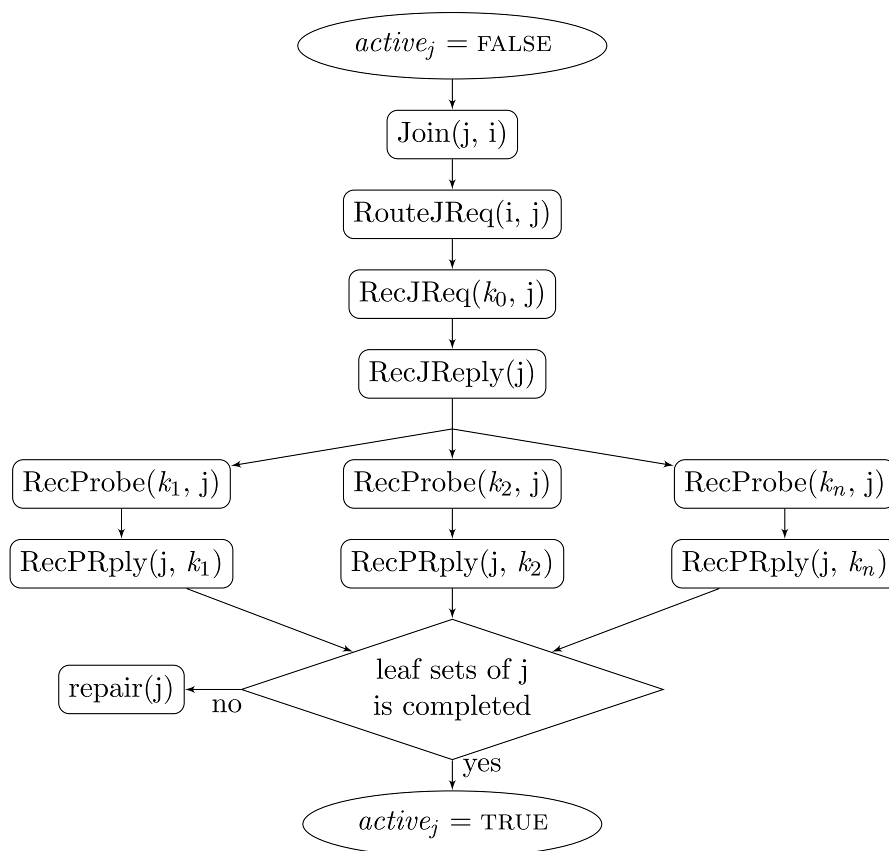

Figure 3.2: Overview of the join protocol of CastroPastry.

action RecJReq(k_0, j), node k_0 replies to j with its current leaf sets to enable node j to construct its own leaf sets. Then in action RecJReply(j), node j *probes* all nodes in its leaf sets in parallel, in order to confirm their presence on the ring. Receiving a "ProbeReply" message, in action RecPRply(j), signals to j that the corresponding node, say k_0, in its leaf sets received the "Probe" message from j in action RecProbe(k_0), and updated its local leaf sets with j. A "ProbeReply" or a "Probe" message contains the updated leaf sets. Each time node j receives a "ProbeReply" message, it updates the local information based on the received message and checks if there are outstanding probes. If no outstanding probe exists anymore or a timeout occurs, it checks whether its leaf sets are *complete*.[1] If so, it finishes the join phase and it status *active* turns to TRUE. Otherwise, it repairs its leaf sets (see action repair(j)) by probing the most distant nodes (leftmost and rightmost) in its leaf sets to get more nodes, retrying to complete its leaf sets.

The system periodically triggers a process of *constrained gossiping* to detect faulty neighbors. In the process, each node will probe nodes in its leaf sets to see if they are still alive. When a node n leaves the network, other neighboring nodes, for example node i, can detect its absence by invoking the function probe(i) periodically. If the probe is not answered within the time threshold, it invokes the action TimeOut(i, j) to repeat probing several times. In case no reply comes back anymore, it starts to repair its leaf sets. Message loss was not mentioned in Castro et al. (2004). Since it is not crucial for the safety property, it is also assumed in CASTROPASTRY that message loss is possible, but no mechanism is especially designed to treat this kind of fault.

3.1.2 Notation

In the following, the algorithm of CASTROPASTRY is illustrated with pseudocode, which is also used as the notation for describing the improved proposal for solving the existing problems of the original version.

The status of a node is described by the Boolean value *active* in CASTROPASTRY based on pseudocode in Castro et al. (2004), whereas intermediate statuses between *active* and $\neg$*active* are introduced later in HAEBERLENPASTRY based on description in Haeberlen et al. (2005) and implementation in FreePastry (2009). The operations on leaf sets and routing tables are mentioned on an abstract level without details. For example, $lset_i.add(\{\dots\})$ means adding a set of nodes into the left and right leaf sets of node i and keeping the closest L nodes on both sides as the new leaf set members. It is important to understand how this operation is implemented in order to understand whether the leaf sets of a node are complete. Therefore, more details in Section 3.2.4 will be illustrated with a formal definition of this operation in Definition 4.1.2. Overloading notation will be used. *rtable* will stand for a function giving the table entry $rtable(r, c)$ in row r and column c. Formulas involving operators $\cup$, $\cap$, $\setminus$ (set minus), $|\dots|$ (cardinality) denote operations on these structures. The notation $\langle$"type"$, \dots\rangle$ denotes a message of some type.

[1] It is not defined in Castro et al. (2004) what it means for a leaf set to be complete. I interpret it as the cardinality of both left and right leaf sets of a node being L.

In the following pseudocode, the imperative commands within the **procedure** block will be executed by node i only when the corresponding prerequisites stated as a boolean expression after the keyword **Require** are fulfilled. Similar to the original pseudocode represented in the paper Castro et al. (2004), here the algorithms can describe *proactive actions*, *(reactive) actions* or *auxiliary functions*. Proactive actions are executed when lookup or join services are required by the Pastry application (e.g. Lookup(i, k) and Join(i, j)), or when Pastry nodes suspect other faulty nodes by periodic probing (e.g. SuspectFaulty(i, j)). (Reactive) actions (e.g. RouteLookup(i)) are executed when a node i receives a message. The auxiliary functions (e.g. probe(i, j)) are the sub-procedures repeatedly called by different actions. They are explicitly described in order to make the pseudocode concise and easy to understand. For simplicity, "send $\langle m, j\rangle$" denotes that the node executing the action sends the message m directly (without routing) to another node j. The function $deliver(i, x, k)$ means a node i directly responds to x's lookup request for k.

3.1.3 Pseudocode of CastroPastry

The overall state variables in the pseudocode of Castro et al. (2004) are *lset* for leaf sets of nodes, *active* for the statuses of nodes, *rtable* for routing tables, *probing* for the set of appending probes of nodes, *failed* for the set of known faulty nodes, and *probe-retries* to remember the times of probing retries.

Note that the TLA$^+$ model CastroPastry of this Pastry model differs in its representations and also does not include *probe-retries*, because retries in the formal model are abstracted as non-deterministic execution of actions. Instead, CastroPastry has a state variable *receivedMsgs* to model the current message pool of the global network.

The following pseudocode represents Pastry model CastroPastry according to the original algorithms from Castro et al. (2004). The formal model of CastroPastry is not shown in the thesis but is available online at VeriDis (2013).

Algorithm 1 initiates the lookup protocol by sending a "Lookup" message to some node i in the network, which node x got from the user operation.

Algorithm 1 Proactive Action: *Lookup* (i, x, k)

Require: Initiated by user operation at the user node x

1: **procedure**
2: send $\langle\langle$"Lookup"$, x, k\rangle, i\rangle$
3: **end procedure**

The action RouteLookup(i, k) in Algorithm 2 calls the auxiliary function *route* to handle a received "Lookup" message.

Algorithm 3 illustrates in detail how node i *routes* the received message m to the node responsible for the key k. Algorithm 4 is an explicit description of a sub-procedure of Algorithm 3, which focuses on how a node *delivers* the message.

In each routing step (see Algorithm 3), a node first checks if the key lies within its leaf sets range, i.e. k lies between the leftmost node and rightmost node of the leaf sets

Algorithm 2 Action: *RouteLookup* (i, k)

Require: Node i receives $\langle$"Lookup"$, x, k\rangle$

1: **procedure**
2: $route(\langle$"Lookup"$, x, k\rangle, k, i)$
3: **end procedure**

Algorithm 3 Auxiliary Function: $route(m, k, i)$ from Castro et al. (2004)

Require: Called by Action RouteLookup(i, j) and RouteJReq(i, j)

1: **procedure**
2: **if** k between $lset_i.leftmost$ and $lset_i.rightmost$ **then**
3: $next$:= pick $j \in lset_i$ such that $|k - j|$ is minimal
4: **else**
5: $r := shared\text{-}prefix\text{-}length(k, i)$
6: $c := r\text{-}th\text{-}digit(k)$
7: **if** $rtable_i(r, c) = null$ **then**
8: $next$:= pick $j \in lset_i \cup rtable_i : |k - j| < |k - i|$
9: $\wedge shared\text{-}prefix\text{-}length(k, j) \geq r$
10: **else**
11: $next := rtable_i(r, c)$
12: **end if**
13: **end if**
14: **if** $next \neq i \wedge next \neq null$ **then**
15: send $\langle m, next\rangle$
16: **else**
17: $receive\text{-}root(m, k, i)$
18: **end if**
19: **end procedure**

Algorithm 4 Auxiliary Function: $receive\text{-}root(\langle$"Lookup"$, x, k\rangle, k, i)$ from Castro et al. (2004)

Require: $active_i$ (Called by Auxiliary Function $route(m, k, i)$)

1: **procedure**
2: $deliver_i(m, k)$
3: **end procedure**

of node i (line 2). In this case, the next hop is to the closest node to the key k chosen from the leaf sets (line 3). Otherwise, it first calculates the length of the prefix match between the key k and node i (line 5). Then, it chooses from R_i, the routing table of node i, the entry in row r and column c (which is returned as a value from the function $r\text{-}th\text{-}digit(k)$ in line 6). The entry in this position has the property that it shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the present node's identifier, because the routing table is constructed specifically to serve this purpose.

If no node is known in this entry ($next = null$ in line 7), the next hop is set to be a node from either routing table or leaf set, whose identifier shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's identifier (line 8). In the end, if the next hop is not the node itself and not *null*, then the message is forwarded to that node, otherwise it executes the procedure *receive-root* (Algorithm 4). In this procedure, the node delivers the message if it is *active*.

When a new node wants to *join* the network it invokes the action Join(j, i) in Algorithm 5, whereby it sends a "JoinRequest" message to some node in the network. The new node initially gets a nearby Pastry node i as given by a user operation.

Algorithm 5 Proactive Action: *Join* (j, i)

Require: Initiated by user operation at node j with given initial node i

1: **procedure**
2: send $\langle\langle$"JoinRequest"$, \{\}, j\rangle, i\rangle$
3: **end procedure**

The node i receives the "JoinRequest" message and routes it to its destination node j (see Algorithm 6) by sending it on the appropriate next hop. The routing continues until the message reaches the nearest node to j. Along the way, each node adds the contents of their routing tables into the message's routing table (line 2 of Algorithm 6).

In response to the "JoinRequest" in action RecJReq(i) in Algorithm 7, the active root node i (the nearest active node to j) adds its own leaf set state for j to the message and returns a "JoinReply". An inactive node receiving such a "JoinRequest" message will simply do nothing and wait until it becomes active to handle the queued messages.

Algorithm 6 Action: *RouteJReq* (i, j)

Require: Node i receives $\langle$"JoinRequest"$, rtable, j\rangle$

1: **procedure**
2: $rtable.add(rtable_i)$
3: $route(\langle$"JoinRequest"$, rtable, j\rangle, j, i)$
4: **end procedure**

The node j receives a "JoinReply" message in action RecJReply(j) in Algorithm 8. It updates its routing table with the node identifiers from the gathered routing table and the leaf sets. Meanwhile, it builds up its leaf sets with the leaf set identifiers from the

Algorithm 7 Auxiliary Function: *receive-root*($\langle$"JoinRequest", $rtable, j\rangle, j, i$)

Require: $active_i$ (Called by Auxiliary Function *route*())

1: **procedure**
2: send $\langle\langle$"JoinReply", $rtable, lset_i\rangle, j\rangle$
3: **end procedure**

"JoinReply" message too, which includes the sender itself as part of its leaf set members. Finally it probes its leaf set nodes .

Algorithm 8 Action: *RecJReply* (j)

Require: Node j receives $\langle$"JoinReply", $rtable, lset\rangle$

1: **procedure**
2: $rtable_j.add(rtable \cup lset)$
3: $lset_j.add(lset)$
4: **for all** $k \in lset_j$ **do**
5: probe(j, k)
6: **end for**
7: **end procedure**

In Algorithm 9, a node probes its leaf set members, which then reply with their leaf sets in action RecProbe(i) (Algorithm 10). Initially, both $probing_j$ and $failed_j$ of a node are empty. They will be extended as soon as probing messages are sent from j or node j notices failure from some nodes, which will be shown later. As soon as i has completed its leaf sets on both sides in action *RecPRply* (Algorithm 11), it has successfully joined the network. Meanwhile, when a node probes its leaf set members, the leaf set members will then probe their leaf set members, and so on. This triggers a chain of probing processes to propagate the updated information (see Algorithms 9, 10, 11).

If the leaf sets are not complete but no new candidate members of the leaf sets are currently being probed (lines 15 and 19 in Algorithm 11), the node invokes function repair(j) (Algorithm 12) to probe its leftmost and rightmost leaf set members in order to get complete leaf sets. In Castro et al. (2004), the protocol implicitly assumes that *there are at least* $L+1$ *"ready" nodes on the ring.* Because otherwise, nodes would never have complete leaf sets and go into deadlock perpetually attempting to repair them.

In the actions RecProbe(i) and RecPRply(i), the failed nodes are also propagated as described in Castro et al. (2004). A node will first check if the received failed nodes in its leaf sets are really faulty by probing them (line 5-7 in both Algorithms 10 and 11) and at the same time remove them from the leaf sets. In this case, the complete leaf sets of an active node might become incomplete, which could lead to a situation of inconsistency. This problem is solved in a later version of the actions RecProbe(i) and RecPRply(i) (Algorithms 20 and 21), where the node does not remove any nodes from its leaf sets when receiving "Probe" or "ProbeReply" messages.

The action SuspectFaulty(i, n) in Algorithm 13 handles the case of node departure. The node i periodically checks the liveness status of its leaf set members by invoking

Algorithm 9 Auxiliary Function: *probe* (j, k)

Require: $k \notin probing_j \wedge k \notin failed_j$ (Called by action *RecJReply*, *RecProbe*, *RecPRply*, *repair* and *SuspectFaulty*)

1: **procedure**
2: send ⟨"Probe", $j, lset_j, failed_j$⟩ to k
3: $probing_j := probing_j \cup \{k\}$
4: **end procedure**

Algorithm 10 Action: *RecProbe* (i)

Require: Node i receives ⟨"Probe", $j, lset, failed$⟩

1: **procedure**
2: $failed_i := failed_i \setminus \{j\}$
3: $lset_i.add(\{j\})$ ▷ here $lset_i$ may be changed
4: $rtable_i.add(\{j\})$
5: **for all** $n \in (lset_i \cap failed)$ **do**
6: probe(i, n)
7: **end for**
8: $lset_i.remove(failed)$
9: $lsetprime := lset_i$
10: $lsetprime.add(lset \setminus failed_i)$
11: **for all** $n \in (lsetprime \setminus lset_i)$ **do**
12: probe(i, n)
13: **end for**
14: send ⟨"ProbeReply", $i, lset_i, failed_i$⟩ to j
15: **end procedure**

Algorithm 11 Action: *RecPRply* (j)

Require: Node j receives $\langle$"ProbeReply", $i, lset, failed\rangle$

1: **procedure**
2: $failed_j := failed_j \setminus \{i\}$
3: $lset_j.add(\{i\})$
4: $rtable_j.add(\{i\})$
5: **for all** $n \in (lset_j \cap failed)$ **do**
6: probe(j, n)
7: **end for**
8: $lset_j.remove(failed)$
9: $lsetprime := lset_j$
10: $lsetprime.add(lset \setminus failed_j)$
11: **for all** $n \in (lsetprime \setminus lset_j)$ **do**
12: probe(j, n)
13: **end for**
14: $probing_j := probing_j \setminus \{i\}$
15: **if** $probing_j = \{\}$ **then**
16: **if** $lset_j.isComplete()$ **then** ▷ both sides of leaf sets have L members
17: $active_j :=$ TRUE
18: $failed_j := \{\}$
19: **else**
20: repair(j)
21: **end if**
22: **end if**
23: **end procedure**

Algorithm 12 Auxiliary Function: *repair* (j)

Require: Called by action *RecPRply*

1: **procedure**
2: **if** $|lset_j.left| < L$ **then**
3: probe$(j, lset_j.leftmost)$
4: **end if**
5: **if** $|lset_j.right| < L$ **then**
6: probe$(j, lset_j.rightmost)$
7: **end if**
8: **end procedure**

the action SuspectFaulty(i, n).

Algorithm 13 Proactive Action: *SuspectFaulty* (i, n)

Require: Initiated by node i periodically

```
1: procedure
2:     probe(i, n)
3: end procedure
```

The action TimeOut(i, j) in Algorithm 14 is initiated when node i does not receive the expected "ProbeReply" message from a particular node j it has probed within some time limit. It first retries the node j several times (*max-probe-retries*). Finally, after the failure of the last trial, node i removes node j from its leaf sets and routing table and invokes Algorithm 12 to repair the leaf sets.

Algorithm 14 Proactive Action: *TimeOut* (i, j)

Require: Initiated by the timer at i for expected "ProbeReply" message from j

```
1: procedure
2:     if probe-retries_i(j) < max-probe-retries then
3:         probe(i, j)
4:         probe-retries_i(j) := probe-retries_i(j) + 1
5:     else
6:         lset_i.remove(j)
7:         rtable_i.remove(j)
8:         failed_i := failed_i ∪ {j}
9:         probing_i := probing_i \ {j}
10:        if probing_i = {} then
11:            if lset_i.isComplete() then
12:                status_i := "ready"
13:                failed_i := {}
14:            else
15:                repair(i)
16:            end if
17:        end if
18:    end if
19: end procedure
```

3.2 The Join Protocol of Pastry

This section starts with a violation of the safety property *CorrectDelivery* of the join protocol in CastroPastry according to Castro et al. (2004). Then, an extension of this join protocol, HaeberlenPastry, is illustrated, which takes its inspiration from the later publication Haeberlen et al. (2005) and the implementation FreePastry (2009).

However, not all the problems are resolved by HaeberlenPastry. Enhancements beyond HaeberlenPastry are explained in the last part of this section.

3.2.1 Concurrent Join Problem and CastroPastry

When verifying the *CorrectDelivery* property, the model checker produced a counterexample, which is illustrated in Figures 3.3 and 3.4, and Table 3.3. In Figure 3.3, the arrows represent neighbor-relationships between node pairs. For example, an arrow from b to c means node b considers node c as its neighbor. According to the calculation of key coverage of nodes, key responsibility is divided equally according to the distance between two neighbor nodes. Therefore, the sector within the circle represents the coverage overlap of nodes a and b (lc_b is the left bound of node b's coverage w.r.t. its left neighbor node c, and rc_a is the right bound of node a's coverage w.r.t. its right neighbor node d). Key k lies within this overlap and it will be shown in what follows how this state of the system is reached from an initial state with just two active nodes c and d (see Figure 3.3) that contain each other in their respective leaf sets (the actual size of the leaf sets being 1). In Table 3.3, each row illustrates the leaf sets' configuration following the simultaneous execution of the actions illustrated in Figure 3.4.

Two nodes a and b concurrently join between nodes c and d. According to their locations on the ring, the join request from node a is handled by node c, and the join request from node b is handled by node d. Both nodes a and b learn about the presence of c and d, and add them to their leaf sets, then send probe requests to both c and d in order to update their leaf sets. Now, suppose that node d is the first node to handle the probe message from node a, and that node c first handles the probe from node b. Learning that a new node has joined, which is closer than the previous entry in the respective leaf sets, nodes c and d update their leaf sets with b and a, respectively (cf. Figure 3.4), and send these updated leaf sets to nodes b and a. Based on the reply from node d, node a will not update its leaf sets because its closest left-hand neighbor is still found to be c, while it learns no new information about the neighborhood to the right. Similarly, node b maintains its leaf sets containing c and d. Now, the other probe messages are handled. Consider node c receiving the probe message from a: it learns of the existence of a new node to its right, which is closer to the one currently in its leaf sets (node b) and therefore updates its leaf sets accordingly, then replies to a. However, node a still does not learn about node b from this reply and its leaf sets remain c and d. In spite of this incorrect information about neighborhood, node a becomes *active*. Symmetrically, node d updates its leaf sets to contain b instead of a, but node b does not learn about the presence of a, even though it also becomes *active*. In the end, the leaf sets of the old nodes c and d are correct, but nodes a and b do not know about each other and have incorrect entries in their leaf sets.

Eventually, a “Lookup” message arrives for key k (see Figure 3.3), which lies between a and b, but closer to a. This lookup message may be routed to node b, which incorrectly believes that it covers key k (since k is closer to b than to c, and b considers c as b's left-hand neighbor), and since it is already *active*, it delivers the key.

This counterexample shows that the join protocol may lead to inconsistent views

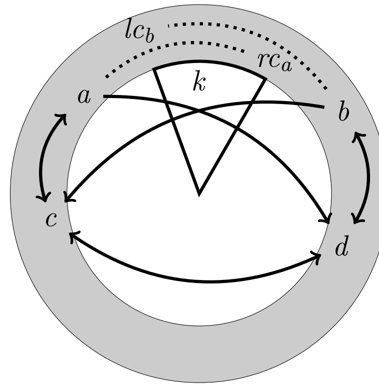

Figure 3.3: The ring configuration for the counterexample.

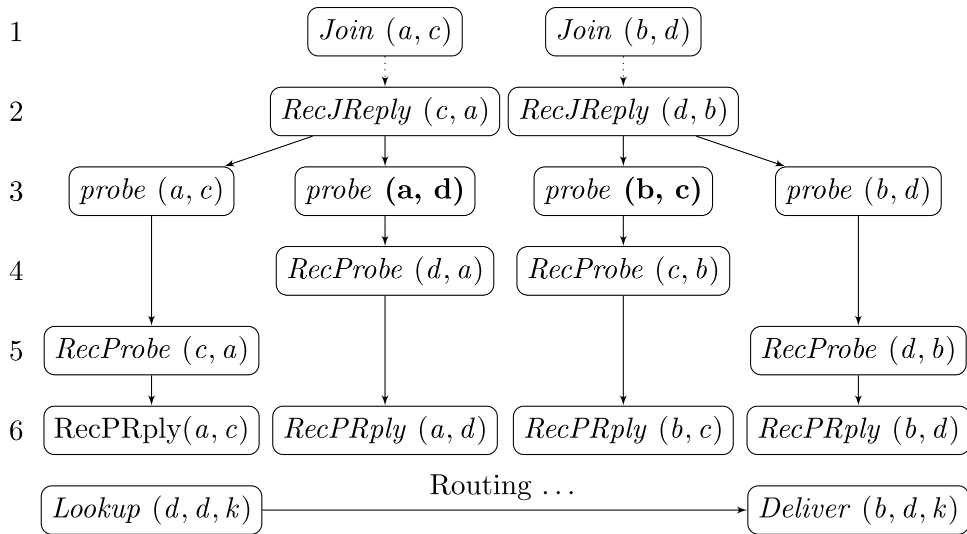

Figure 3.4: Counterexample leading to a violation of *CorrectDelivery*.

Row	Action	$lset_a$	$lset_b$	$lset_c$	$lset_d$
1	*Join* (a, c) || *Join* (b, d)	N/A	N/A	{d}	{c}
2	*RecJReply* (c, a) || *RecJReply* (d, b)	**{c, d}**	**{c, d}**	{d}	{c}
3	*probe* (a, c) || ... || *probe* (b, d)	{c, d}	{c, d}	{d, **b**}	{c, **a**}
4	*RecProbe* (d, a) || *RecProbe* (c, b)	{c, d}	{c, d}	{d, b}	{c, a}
5	*RecProbe* (c, a) || *RecProbe* (d, b)	{c, d}	{c, d}	{d, **a**}	{c, **b**}
6	*RecPRply* (a, c) || ... || *RecPRply* (b, d)	{c, d}	{c, d}	{d, a}	{c, b}

Table 3.3: Collision analysis of the leaf sets

by active nodes of their neighborhoods on the ring, and therefore the protocol does not guarantee *CorrectDelivery*.

3.2.2 Lease-Granting Protocol and HaeberlenPastry

Indeed, following the initial publication of Pastry, Haeberlen et al. (2005) presented a refined description of Pastry's join protocol, which repairs the problem illustrated in the counterexample shown above. This improvement is modeled as HAEBERLENPASTRY.

In contrast to the join protocol in CASTROPASTRY, described in Section 3.1.1, the refined join protocol in HAEBERLENPASTRY requires an explicit "transfer of coverage" from the active neighbor nodes before a joining node can become active and answer lookup requests, which will be explained in what follows.

In fact, HAEBERLENPASTRY is also inspired by the implementation in FreePastry (2009), where leases expire periodically and are requested from the neighboring nodes. HAEBERLENPASTRY is not illustrated with pseudocode here, but its TLA$^+$ code can be found on the Web in VeriDis (2013). Instead, a final version of Pastry, LUPASTRY, will be shown later in Section 3.3.3 with pseudocode.

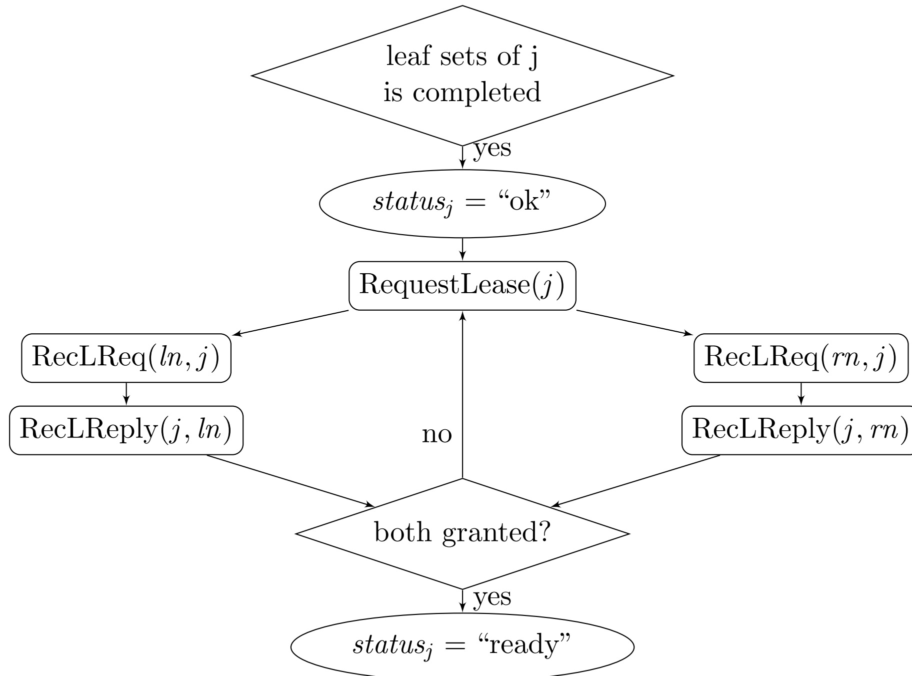

Figure 3.5: Extending the join protocol by lease granting.

Figure 3.5 depicts the extension to the join protocol as described in Section 3.1.1 (compare Figure 3.2). After node j has built complete leaf sets, it reaches an intermediate status between *active* and *inactive*, which is modeled as a status "ok" in action *RecPRply* (see also Algorithm 21). As soon as it becomes "ok", it starts to request the leases from

its direct neighbors by sending “LeaseRequest” messages to its neighbors *ln* and *rn* (the two closest nodes in its current leaf sets), requesting a lease for the keys it covers. A node receiving a lease request from another node that it considers to be its neighbor grants the lease (see also Algorithm 23). Only when both neighbors grant the lease will node j become *active*, modeled as “ready” in this and later models of Pastry (see also Algorithm 24 for details).

From here on, state variable *active* is replaced with *status*, in order to maintain a compatible notation of the statuses of a node. There are four different statuses of a node: “dead”, “waiting”, “ok” and “ready”, where “dead” stands for inactive nodes as before, “ready” stands for active nodes as before, “ok” is the intermediate status introduced by HaeberlenPastry, and “waiting” is introduced to fill the gap for formal verification purposes, to describe a node’s status when it starts to join and can route messages, but is not yet “ok”. More details of statuses will be explained later in Section 3.3.2, when the final model of Pastry is introduced.

The node j could have incomplete leases from both of its neighbors due to different reasons:

- The “LeaseReply” message may still be on its way (delayed) or lost. In this case, node i could invoke the action *RequestLease* after some time limit. to resend the “LeaseRequest” message.
- The direct neighbor of node j considers some other freshly joined node as its direct neighbor and therefore declines to grant the lease. In this case, the inconsistent view of the neighborhood needs to be first fixed by having node j or its neighbors invoke the action *SuspectFaulty* to probe each other and detect new nodes. Then node j can send a “LeaseRequest” message to the correct direct neighbor, which will then grant the leases. The “LeaseRequest” messages to the wrong neighbors during the probing process will simply be ignored in action *RecLReq*.

In HaeberlenPastry, departure of nodes is handled together with the lease granting mechanism (see Figure 3.6). Starting from the top right of Figure 3.6, a “ready” node periodically requests refreshed leases from its direct left and right neighbor, in order to ensure the correctness of its neighborhood information. When a lease of one of the neighbors (*ln* stands for left neighbor, *rn* stands for right neighbor in Figure 3.6) expires, the node goes back to the status “ok” and requires a renewal of its lease from that neighbor. When a node j detects another node n to be faulty and node j has probed the node n several times without any success, then node j does not remove node n from its leaf sets at once, neither does it report this failed node to its neighbors. Unlike in Algorithm 14 (here a star * is added after the action name *TimeOut* in Figure 3.6), it first checks if it still holds the *grant* of that node, in order to know if the lease of n has expired. If not, it only marks node n as *failed* by adding it into the variable $failed_j$, waits until the local $grant_j$ of the node n expires, which must happen after node n has lost its lease of node j. Then node j declares node n to be “dead” and removes it from its local leaf sets. After removal, it will try to repair its leaf sets by probing its farthest remaining leaf set members to complete the leaf sets.

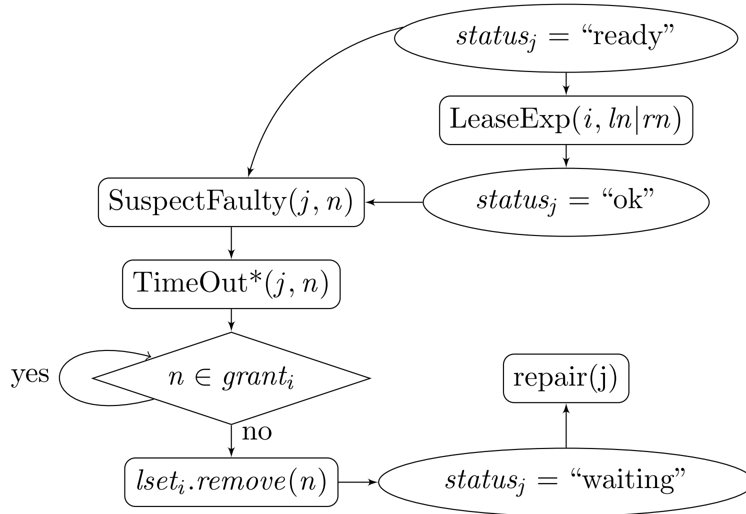

Figure 3.6: Node departure handling.

3.2.3 Related Work on Pastry

Although the Pastry version published in Castro et al. (2004) and the later publication Haeberlen et al. (2005) are the major focus of this thesis, relevant publications about Pastry are also explored to gain deeper understanding of the protocol.

The initial idea of the Pastry routing algorithm was published in Rowstron and Druschel (2001), where the basic idea of using a virtual ring overlay is introduced. More details are illustrated later in Castro et al. (2003). In addition, the paper Rowstron and Druschel (2001) also illustrates how the routing can be performed efficiently, in a logarithmic number of steps around a ring structure. This aspect is then explained in detail in a later publication Caesar et al. (2006). Furthermore, the paper Rowstron and Druschel (2001) introduced the leaf sets structure for a network node to maintain a consistent local view of its neighboring nodes w.r.t. its Pastry identifier. The purpose of this kind of data structure is to provide dependable routing; in other words, to provide correct routing and prevent "routing anomalies". A later publication Castro et al. (2004), focuses more on dependability of routing.

As a first verification approach on such a real world complex overlay network protocol, the focus has been set to verifying its correctness property, rather than its performance property. On the one hand, the evaluation results shown in Castro et al. (2003) and Castro et al. (2004) for its performance are already promising. Besides, the adoption of this kind of network protocol in various peer-to-peer applications affords a high degree of confidence in its performance. On the other hand, applying formal methods to problems relating to small distributed system is already a non-trivial research topic in its own right. It is challenging enough to verify the correctness of such network protocols with dynamic membership, even when real time and performance issues are abstracted away.

Focusing on the correctness property, the approach starts with the most rigorous description of Pastry available to date which is the pseudocode illustrated in Castro et al. (2004). This paper focuses more on the *leaf sets* data structure, which is called a *virtual neighbor set* in Caesar et al. (2006). It will emerge later that this data structure plays a dominant role of the correctness property of Pastry.

After modeling Pastry in TLA$^+$ as CastroPastry and employing the TLC model checker to find bugs, several problems with the consistency of Pastry were discovered as illustrated later in Section 3.2.4. With all these open questions, discussions were held with one of Pastry's original designers, and another publication Haeberlen et al. (2005) with many improvements towards consistency was pointed out.

To ensure greater security against path failure, in particular asymmetric connectivity among nodes and network partition, the paper Haeberlen et al. (2005) has given a more detailed description of the leaf sets. It assumes that each pair of leaf set members is connected. Under this assumption, *virtual links* and *route recovery* are introduced to update the one-way routing path from a node to its local leaf set nodes. Since the routing path mechanism is beyond the scope of this thesis, and is not of the essence of virtual ring overlay network protocols, no assumptions on the paths are made for the verified Pastry model. The only interesting improvement relevant to the verification task of this thesis is how the routing mechanism deals with the problems which arise when one node has just joined the network while the other nodes are not yet aware of its presence. The techniques used in Haeberlen et al. (2005) are periodic probing, maintaining a different liveness status for each node locally in their leaf sets, and adding an additional key coverage transfer protocol after a node has completed its join protocol.

The source code of FreePastry (2009) is also explored to see if and how different techniques mentioned in Castro et al. (2004) and Haeberlen et al. (2005) are implemented in real systems. In fact, the liveness check and key coverage agreement are implemented by a periodic lease granting mechanism with expiring leases which represent how long the current network node believes its local liveness statuses of other nodes. The periodic lease requesting protocol is modeled with non-determinism as the enabling condition of a TLA$^+$ action, as formally illustrated in Definition 4.2.13 in Section 4.2.

3.2.4 Enhancements of the Protocol in IdealPastry and LuPastry

The lease granting protocol in HaeberlenPastry solves the problem described in Section 3.2.1. But as discovered later during our theorem proving process, there is a more efficient solution to the concurrent join problem shown before; moreover, there are other problems that will be illustrated in this section, which the lease-granting protocol cannot solve. In the following some of the problems and their solutions will be demonstrated on an abstract level. More details about how the problems were detected will be illustrated in Chapter 5.

Unclear Completeness of Leaf Set

In both CastroPastry and HaeberlenPastry, checking the completeness of leaf sets is a crucial step before a node upgrades its status. But after gaining a deeper understanding of the protocol through formal verification, another problem occurred: it turns out that it is not trivial for a node to decide whether its leaf sets are complete or not, due to the fact that its left and right leaf sets might overlap even though the cardinality of each set is already L, which is the maximal length of each leaf set. Furthermore, a node might never upgrade its status at this step if less than L nodes are available (i.e. not "dead") on the entire ring.

To solve these two problems, checking the length of the leaf sets is abandoned as a precondition for a status upgrade in IdealPastry and LuPastry. Instead, whether the variable *probing* is empty becomes the sole focus of the check. This change makes sense because on one hand, if *probing* is empty, all expected probing messages have been answered and the node has no further new nodes to probe. Then the node should either already have enough nodes in both its leaf sets, or there are less than L active nodes in total on the ring. In both cases, the node should upgrade its status. On the other hand, a node does need to check if there are any new candidates for its leaf sets before it upgrades its status, even if it has already collected a sufficient number of them, in order to ensure a more consistent local view of its neighborhood. This change is also preferable because checking the emptiness of the variable *probing* relies only on the information a node could obtain from the protocol, whereas checking completeness of leaf sets makes the join protocol more vulnerable to the number of active nodes on the ring and the parameter L.

Updated vs. Previous Leaf Set

Another possible solution to the concurrent join problem shown in Section 3.2.1 is to reply with the previous leaf sets instead of the updated leaf sets (see Algorithm 10) before adding the probing node. This improvement prevents the event that an incoming "Probe" message should cause the recipient to remove a neighbor from its local leaf set, and then also omit that neighbor in its reply to sender. Details of the solution are described in Algorithm 20, which is also model checked (in Chapter 5) without any further violation caused by the concurrent join problem. Hence this improvement is adopted in IdealPastry and LuPastry.

Since the problem can be solved efficiently without the lease granting protocol as shown above in Section 3.2.1, intuitively one would like to optimize away altogether the lease granting protocol and the intermediate "ok" status. The following Section, 3.2.4, shows that using the lease granting protocol to distinguish the status "ok" from "ready" is crucial, because otherwise an "ok" (likewise a "ready") node which has exhausted all probings (i.e. it has no new node to probe) does not always have a key mapping consistent with other such nodes. For this reason, the lease granting protocol is preserved in IdealPastry and LuPastry.

The Ignored "ok" Node

In Figure 3.7, there are four different nodes involved. Two nodes (kl, kr) try to join between other "ready" nodes l and r. The node kl is closer to l and node kr is closer to r. Assume that kl probes r after it got a "JoinReply" from node l, and then kr tries to join through r. Then node kr might probe kl and r, becoming "ok", while l still believes r is its direct right neighbor. Now if node kr could skip the lease granting protocol and directly become "ready", it would have coverage conflict with node l for some region between kl and kr.

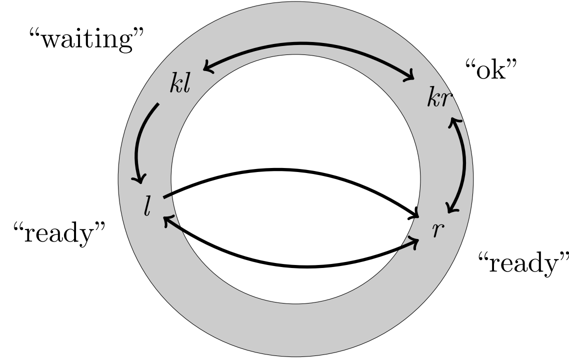

Figure 3.7: The ignored "ok" node.

It is obvious that if kr has to run the lease granting protocol, it will never become "ready" before node kl has become "ok", as this relies on the "ProbeReply" message from l to kl, which is sent only after l has handled the "Probe" message from kl. By that time, kl should have replaced r in the right neighbors leaf set of l to prevent the case demonstrated in Figure 3.7.

Rejoin Problem

When model checking the HaeberlenPastry with its mechanism for recovering from node departure illustrated in Figure 3.6, the model checker TLC discovered a counterexample. A short explanation will be provided here and more explanations will be given in Chapter 5.

Initially, only node b is ready and all the other three nodes are dead. Nodes a, c and d concurrently join around b. The "JoinReply" message sent from b to d is delayed, while nodes a and c successfully stabilize their states via the consistent join protocol and lease granting protocol. After the 3 nodes (a, b, c) have connected to each other, node b leaves the network silently. By this time, the delayed message from the previously "ready" node b to the awaiting node d is received by node d, leading node d to believe that b was the only "ready" node in the network. So node d probes b. Node b now rejoins the network by sending a "JoinRequest" message to node a. Then, node b receives the probing message from d and replies to it because its status is now "waiting". The node d receives the "ProbeReply" message and hence becomes "ok". Till now, none

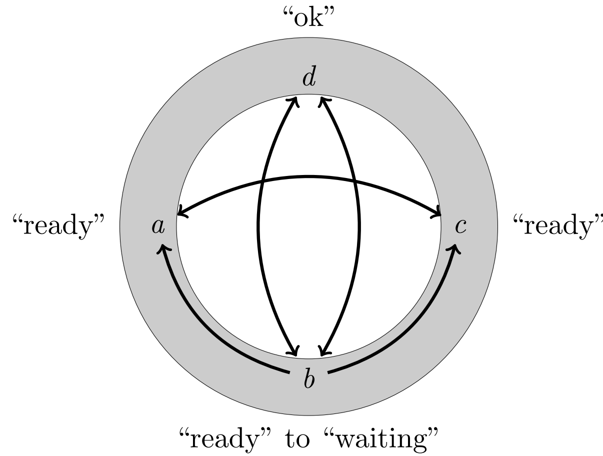

Figure 3.8: Rejoin counterexample.

of the direct "ready" neighbors of d (nodes a and c) are aware of the node d. In case message delay is allowed, b might no longer try to wait for a probe reply from a and starts probing d to repair its leaf sets and these two nodes will stabilize themselves and become "ready". Hence, there is a separation in the network.

In this counterexample, a joining node waiting for a "JoinReply" message has stopped waiting upon receiving a "Probe" message, which leads to completion of the join process. A final fix in IDEALPASTRY and LUPASTRY to prevent this case is to add a precondition for handling any incoming "Probe" message, that only "ready" nodes or nodes with non-empty leaf sets can receive a "Probe" message, as illustrated in Algorithm 20.

The problems discussed above are found already with four nodes involved in the network, more problems will occur when more nodes are involved as demonstrated in the following examples.

Separation of Networks Due to Departure of Nodes

In order to relax the assumption that nodes do not leave the network, possible violations of *CorrectDelivery* is analyzed, which leads to the following counterexample and motivates the preservation of this strong assumption.

Starting from the initial state that all nodes (a, b, c, d, e and f) are "ready", suppose that two particular nodes a and d simultaneously leave the network. The neighboring nodes detect faults by invoking the action SuspectFaulty(i, j) and when the leases expire, use action TimeOut(i, j) to remove them from their leaf sets.

Subsequently, node c probes the only left leaf set member b and gets the "ProbeReply" message to become "ok". Respectively, node b probes c and also gets a "ProbeReply" message from c, such that it believes b and c are the only nodes on the network and also becomes "ok". They can then grant each other and become "ready". Simultaneously,

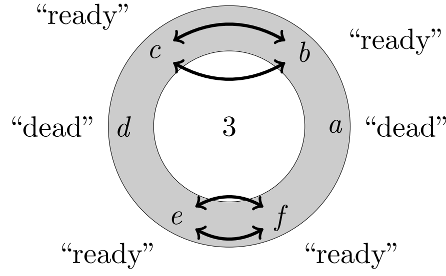

Figure 3.9: Separation of the network due to simultaneous departures of nodes.

nodes e and f complete the same process to build a separate network as illustrated in Figure 3.9.

Hence, two separate networks exist on the ring, such that neither is aware of the other. When a "Lookup" message comes to nodes c and e for looking up the key d, both of the actions Deliver(c, d) and Deliver(e, d) are enabled to deliver the "Lookup" message.

This counterexample illustrates how network separation occurs when nodes in certain positions leave the network simultaneously, which is also mentioned in Haeberlen et al. (2005). Since no solution is known which solves this problem thoroughly, the repair protocols are no longer included among the possible actions as targets of verification. Moreover, this counterexample motivates the assumption that no nodes leave the network (Assumption 3.4.3) in LuPastry.

More Concurrent Join Nodes

As Section 3.2.2 introduced, the lease granting protocol of HaeberlenPastry does not place any restriction on how many joining nodes a "ready" node can take care of. Since the assumptions of IdealPastry disallow concurrent joins among nearby "ready" nodes, this is not a problem. But as this assumption is relaxed in LuPastry, the following counterexample will illustrate that the safety property *CorrectDelivery* can be violated when more than two nodes concurrently join between neighboring "ready" nodes, even without any node departure.

Assume the network contains five nodes l, s, ln, i, rn as shown in Figure 3.10. Both nodes l and rn are initially "ready" nodes. All the rest are initially "dead". The leaf set length L is one, meaning each node has only one node in each of its left and right leaf sets.

First, node s joined the network through node rn and received a "JoinReply" message from node rn. But the probing message sent out from node s to node l was delayed: only node rn received the "Probe" message from s. Node rn now has a new left neighbor, node s. Then node ln joined the network through rn. It got a "JoinReply" message from node rn containing node s. It probed both node s and node rn and became "ok" after nodes s and rn replied to both "Probe" messages. Note that node s can reply to such "Probe" messages when it is in state "waiting". Now node i joins the network

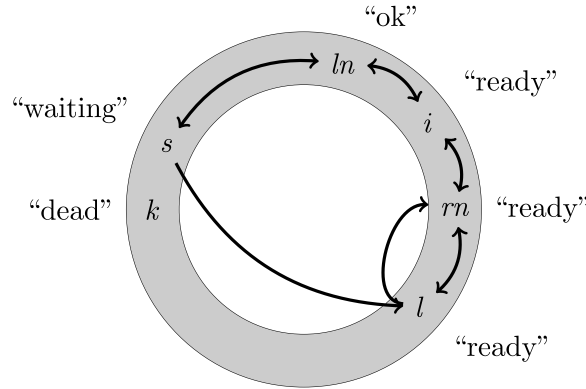

Figure 3.10: Concurrent join with 5 nodes.

through *rn*. It got a "JoinReply" message from node *rn* containing node *ln*. Then node *i* probed nodes *ln* and *rn* and became "ok". After that, node *i* requested the leases from both its direct neighbors *ln* and *rn*. Finally, node *i* receives leases from both sides and becomes "ready".

Now node *l* still believes that there is only one "ready" node *rn* as its left and right neighbor. There is a region covered by both *ln* and *l*. If then a "Lookup" message is sent to node *l*, which looks up the key *k* within this region, then both node *l* and node *ln* can deliver it.

In this counterexample, no node has left the network and only pure joining of nodes is allowed. In conclusion, HaeberlenPastry cannot ensure the safety property *CorrectDelivery* even without departure of nodes.

This problem is solved in LuPastry by restricting the number of concurrent joining nodes that any "ready" node may allow at any given time. To implement this restriction, the *toj* variable is introduced in the solution (see Algorithm 18), to keep track of whether a "ready" node has replied to the "JoinRequest" message of a joining node, in order to make sure that it can help at most one node to join at a time. If the "ready" node does not have such a joining node assigned to it, this variable has the value of the "ready" node's own identifier.

A final solution of this problem is to allow nodes to add joining nodes to their leaf sets on receiving a "JoinRequest". The node should be in charge of only one node which it is currently letting join. More detailed illustration of this kind of problem will be shown in Chapter 5.

3.3 The Verified Pastry Join Protocol LuPastry

Starting from the analysis and discussion in Section 3.2.4, an improved consistent join protocol IdealPastry has been developed and verified, assuming that no node leaves the network and no concurrent join occurs in the region of nearby "ready" nodes. Details of IdealPastry are explained in Section 6.3. After that, the assumptions are further

relaxed to no departure of nodes and the protocol is improved to LuPastry.

This Section focuses on the introduction of the final version LuPastry. The parts handling node departure are omitted, because only the verified protocol is demonstrated and those parts of the protocol are not verified due to Assumption 3.4.3 that no node leaves the network. The reason for making such an assumption is a long story, which will be explained in Section 3.4. In the following, this final version of Pastry with all improvements will be illustrated, much like the algorithms shown in Section 3.1.1. This protocol can be directly implemented in any programming language. Later in Section 4, this protocol will be described with more details in a formal language TLA^{+}, to which the verification approach in Chapter 6 will refer.

3.3.1 Introduction with Flow Chart and Running Example

Here the join process of the improved Pastry will be illustrated as a flow chart in Figure 3.11. The whole join process consists of three main steps: from "waiting" to "ok", from "ok" to "ready", and the last announcement of being "ready".

- The first step starts from the initial state, when a node is "waiting" and starts to join the network. It describes how a node upgrades from "waiting" to "ok". This step implements the consistent join protocol described in CastroPastry, which has already been illustrated in the flow chart shown in Figure 3.2. In this flow chart, the state variable *active* is replaced with *status* to maintain compatibility with HaeberlenPastry.

- The second step starts from the "ok" status of a node, and describes how it upgrades to "ready". This step implements the lease granting protocol developed by HaeberlenPastry, which is also demonstrated in the flow chart shown in Figure 3.5.

- In the last step, the newly joined node is completely "ready", meaning fully-functional in the sense of being capable of of executing all the actions other "ready" nodes can perform. But it still needs to grant leases to its left and right neighbors in order to inform them that it has become "ready" so that one of the neighbors who has been helping it to join and blocking all other join requests can now release the block and process further join requests.

In this final big picture, the relevant protocols for handling node departure illustrated in Figure 3.6 are not included, because in the final verification protocol, node departure is excluded by a strong system assumption. It is explained later in Chapter 6 how node departure will destroy the desired property *CorrectDelivery* and why it is not possible to work around this problem.

To better understand the complete join process of a node, a running example will now be introduced (compare Table 3.4 and the sequence of the figures illustrated below).

In the running example, the routing table is ignored because it does not have any effect on joining, as can be seen from Chapter 6. Therefore, only the variables with altered values are demonstrated here. The variable *toj* is introduced as a solution to

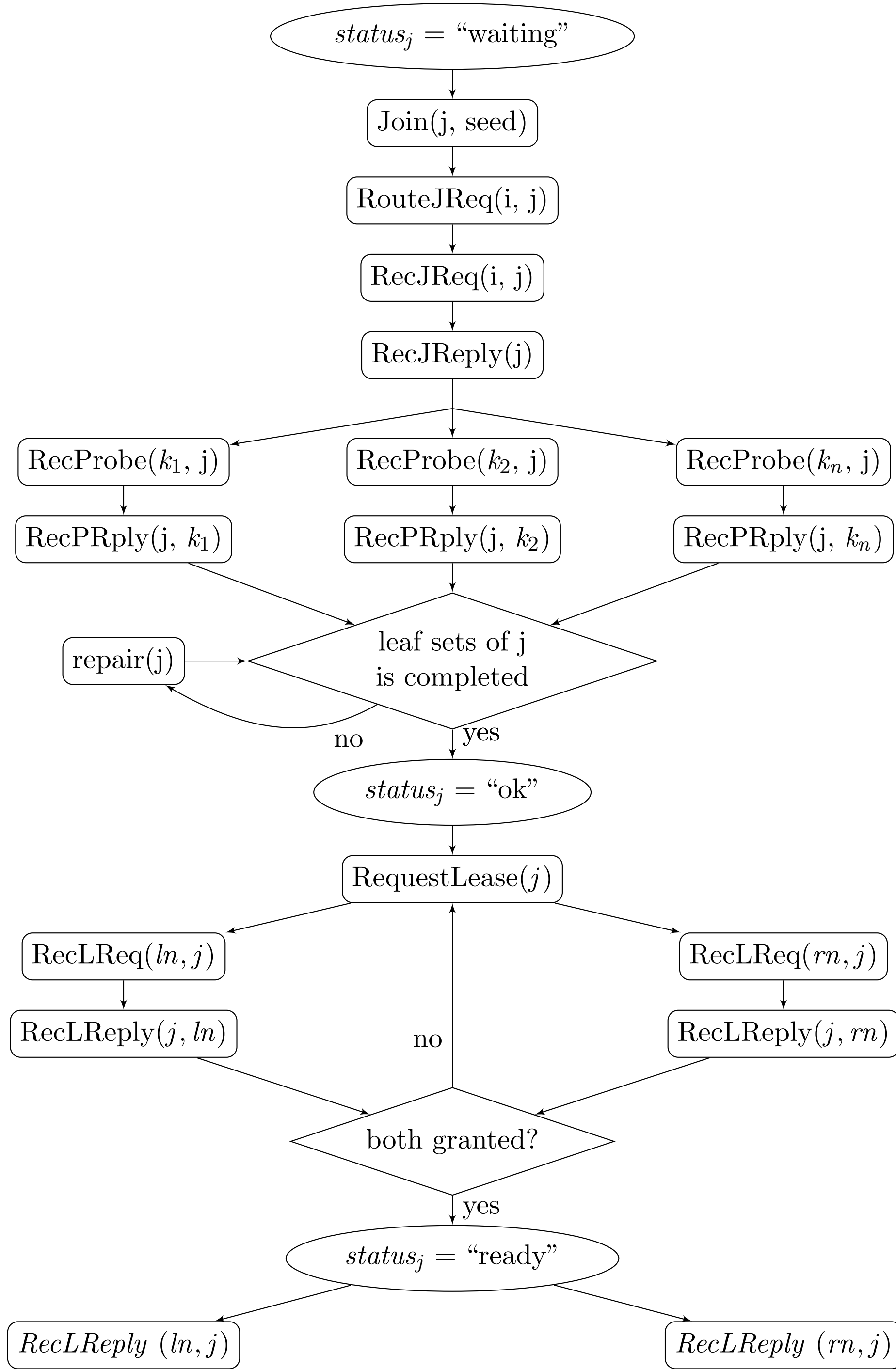

Figure 3.11: Flow chart of complete join process of LuPastry.

the concurrent join problem with five nodes illustrated in Figure 3.10. It is used to keep track of whether a "ready" node has replied to any "JoinRequest" message (from a joining node).

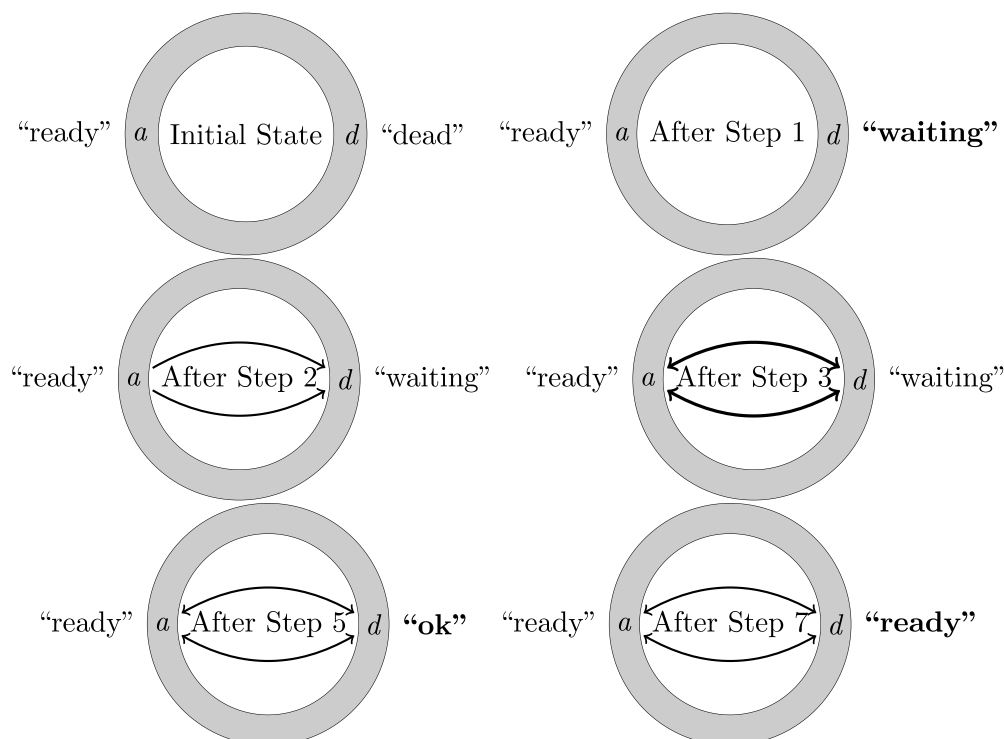

Figure 3.12: Join example: upgrades of the status.

Initially there is a single "ready" node a on the ring at Step 0 shown in Figure 3.12. According to the assumption that a "ready" node can handle at most one joining node at a time (derived by consideration of the concurrent join problem in the previous Section), only one new node is able to join. In the following it will be illustrated how node d can go through different statuses ("waiting", "ok", "ready") and finally join the network. Meanwhile, it is also shown how the status of a node affects its behavior.

Step 0 Initially as shown in Figure 3.12, node a is "ready" and d is "dead". The leaf sets and probing sets are all empty. Their toj_i point to themselves; their grants and leases contain only themselves.

Step 1 The new node d joins by executing the action Join(d, a).

Step 2 Node a executes the action RecJReq(a). Routing is not applicable here because there are no other nodes to route to and node a covers the whole ring.

It adds d into its leaf sets. Now its left and right neighbors are both node d. It has now a node to join ($toj_a := d$) and sends a "JoinReply" message back to node d containing its own previous leaf sets (the empty leaf sets).

Step	Actions	$lset_a$	$lset_d$	$probing_d$	$grant_a$	$grant_d$	$lease_a$	$lease_d$	toj_a
0		{}	{}	{}	{a}	{d}	{a}	{d}	a
1	*Join* (d, a)	{}	{}	{}	{a}	{d}	{a}	{d}	a
2	*RecJReq* (a, d)	**{d}**	{}	{}	{a}	{d}	{a}	{d}	d
3	*RecJReply* (d, a)	{d}	**{a}**	**{a}**	{a}	{d}	{a}	{d}	d
4	*RecProbe* (a, d)	{d}	{a}	{a}	{a}	{d}	{a}	{d}	d
5	*RecPRply* (d, a)	{d}	{a}	**{}**	{a}	{d}	{a}	{d}	d
6	*RecLReq* (a, d)	{d}	{a}	{}	**{a, d}**	{d}	{a}	{d}	d
7	*RecLReply* (d, a)	{d}	{a}	{}	{a, d}	**{a, d}**	{a}	**{a, d}**	d
8	*RecLReply* (a, d)	{d}	{a}	{}	{a, d}	{a, d}	**{a, d}**	{a, d}	**a**

Table 3.4: State variables in the join example

Step 3 The node d executes action RecJReply(d).

It initiates its leaf sets with a as both its left and right neighbor. It probes a and adds a to the probing set ($probing_d := \{\} \cup \{a\}$).

Step 4 The node a receives a "Probe" message from node d as precondition of action RecProbe(a) and replies.

Step 5 Node d receives the reply from a's reply in action RecPRply(d).

It deletes node a from its probing set, then it has no nodes to probe, and hence becomes "ok". Then node d sends a "LeaseRequest" to its left and right neighbors: node a.

Step 6 Node a grants the lease in action RecLReq(a) (Algorithm 23).

Node a adds node d into its granted nodes ($grant_a = \{a, d\}$) and sends a "LeaseReply" message back to d containing local leaf sets and a granting bit (*isNeighbor* = TRUE).

Step 7 Node d receives the "LeaseReply" from node a in action RecLReply(d).

It adds node a into its set of leases ($lease_d = \{a, d\}$) and then becomes "ready". From now on, the newly joined node d can deliver and answer "JoinRequest" messages from other new nodes. But node a cannot reply to "JoinRequest" messages. Therefore, node d sends a "LeaseReply" to both of its neighbors (a) with a granting bit (*grant* = TRUE) and adds the neighbors to its set of granted nodes ($grant_d = \{a, d\}$).

Step 8 Node a receives a "LeaseReply" from node d in action RecLReply(a).

It adds d into its set of leases ($lease_a = \{a, d\}$). It now sets the (state) variable toj_a back to itself. From now on, the "ready" node a can again answer "JoinRequest" messages from other nodes.

3.3.2 Statuses of a Node

Together with the illustrated workflow in Section 3.3.1, I give more details of the statuses of a node here: their prerequisites, applicable actions (compare the definitions of the actions above) and invariants.

- The “dead” nodes
  - All of its keys can be interpreted as “dead”. It turns from “dead” to “waiting” when it is assigned to a node to join the network in action Join(j, i).
  - As soon as a node leaves the network, its status turns to “dead”.
  - A “dead” node can execute the action Join(j, i).
  - All dead nodes have empty leaf sets.
- The “waiting” nodes
  - A node can only becomes “waiting” in action Join(j, i) (compare Algorithm 15).
  - As soon as a node has no more candidates in its leaf sets to probe and all outstanding “Probe” messages have been answered, it turns from “waiting” to “ok” in action RecPRply(j). It turns from “waiting” to “dead” when it leaves the network.
  - During the time a node is “waiting”, it can handle “JoinReply” messages in action RecJReply(i); it answers “Probe” messages in action RecProbe(i);or handles “ProbeReply” messages in action RecPRply(i);
  - For the “waiting” nodes we can state the following invariants:
    * If there is only one “ready” or “ok” node i in the network, then all the “waiting” nodes can only have themselves and node i in their leaf sets.
    * The sender of any “JoinRequest” message must be of status “waiting”.
    * The destination of any “JoinReply” message must be of status “waiting”.
- The “ok” nodes
  - The status “ok” is the next stage after “waiting” in the consistent join protocol. A node turns to “ok” under the conditions described above in action RecPRply(i).
  - A node turns from “ok” to “ready” when it receives leases from both its left and right neighbors in action RecLReply(i). It turns from “ok” to “dead” when it leaves the network.
  - Similar to “waiting” nodes, an “ok” node can answer a “Probe” messages in action RecProbe(i) and handle “ProbeReply” messages in action RecPRply(i). It can also request leases from its direct neighbors in action RequestLease(i), answer a “LeaseRequest” message in action RecLReq(i) or handle a “LeaseReply” message in action RecLReply(i).

  - For the nodes which are at the "ok" or "ready" stage of the joining process we can state the following invariants:
    * If a node i has granted another node j, then both of them must be either "ok" or "ready".
    * The sender of "LeaseReply" messages is either "ok" or "ready".
    * The sender of "LeaseRequest" messages is either "ok" or "ready".
    * If there is more than one "ok" or "ready" node in the network, then they must have different neighbors (meaning their leaf sets are not empty). In case there is only one node in the network that is either "ok" or "ready", then that node must have empty leaf sets, all the "waiting" nodes will have that node in their leaf sets. In that case no "ProbeReply" or "LeaseReply" can occur. If there is a "Probe" message, its destination must be that "ok" or "ready" node. If there is a "JoinReply" message, the leaf sets sent within the message must be empty.

- The "ready" nodes
  - A node can only become "ready" in action *RecLReply*,by which an "ok" node receives leases from both its left and right neighbors.
  - A "ready" node can execute all the actions that "ok" nodes can execute. Furthermore, it can deliver a "Lookup" messages in action RouteLookup(i) and answer a "JoinRequest" messages in action RecJReq(i).
  - The following invariants are relevant only for "ready" nodes:
    * There is no "ready" node between a "ready" node and its immediate neighbor.
    * Only "ready" nodes can let other nodes join.
    * If a "ready" node i covers the identifier of another node k, then no other node j has allowed that node k to join through j.
    * Given a "JoinReply" message to node j, there can never be a "ready" node between the initial left and right neighbors of node j, after it has used the received leaf sets to built up its own leaf sets.
    * If a "LeaseReply" message is sent by a node that has joined through the destination of the "LeaseReply" message, then the sender must be "ready".
    * The sender of a "JoinReply" message must be "ready".

3.3.3 Pseudocode of LuPastry

The same variables *lset*, *rtable*, *probing* and *failed* are used as in Section 3.1.3. The variable *active* is replaced with *status* as introduced in Section 3.2.2 and explained in Section 3.3.2. New variables *lease* and *grant* were introduced in connection with the

lease-granting protocol in Section 3.2.2. The variable *toj* was introduced as a solution to the concurrent join problem in Section 3.2.4.

The pseudocode for the action Lookup(i, k) in LuPastry is omitted because it is exactly the same as in Algorithm 1 for CastroPastry. In contrast with the action Join(i, j) in CastroPastry shown in Algorithm 5, the new action in LuPastry in Algorithm 15 ensures that only "dead" nodes can join, which prevents rejoining by nodes which have already started the joining process. With the exception of this action, all other actions may only be executed by nodes which are not "dead". Since node j was already in the network and should be publicly known, it should be a "ready" node.

Algorithm 15 Proactive Action: Join(i, j)

Require: $status_i$ = "dead" $\wedge j$ = "ready"

1: **procedure**
2: $status_i$:= "waiting"
3: send $\langle\langle$"JoinRequest", $\{\}, i\rangle, j\rangle$
4: **end procedure**

Lookup Request Handling

Unlike the same action RouteLookup(i) in CastroPastry described in Algorithms 2, 3 and 4, more preconditions must be met before a node can handle the "Lookup" request in Algorithm 16. For example it should be explicitly written in a formal model that "dead" nodes do not react to any message.

Here a node delivers a "Lookup" message when it is "ready" and *covers* (see Section 3.1.1) the key j, which means that j is closer to i than any other member in $lset_i$. If it does not cover j, it searches for a candidate closer to j for its next hop by invoking the function *findNext* (i, j). In case no such node exists, it reports an error to the user application.

Besides, there is an explicit separation of correct key delivery and faulty delivery (reportError $\langle$"NoLegalRoute", $i, j\rangle$), such that if a node delivers a key, it must cover it. This separation makes it possible to reduce the property *CorrectDelivery* to properties of correct coverage of leaf sets in the later approach to verification.

The precondition of action RouteLookup(i, j) shown in Algorithm 16 invokes the function $covers(lset_i, j)$, to decide if node i can deliver it or has to forward the message to another node. In the latter case, if it finds the next hop, then it will forward the "Lookup" message to that node. Otherwise (if the returned value is the identifier of the node itself) it will send a "NoLegalRoute" message to report a routing error.

The error reporting case can never occur as long as the function findNext(i, j) and $covers(lset_i, j)$ are correctly implemented. Intuitively, $covers(lset_i, j)$ is only FALSE when one of the neighbors of node i, say the left neighbor *ln*, is closer to the key j than i itself. The function findNext(i, j) as implemented in Algorithm 17 is designed to find a closer node, at worst *ln*, to forward the message, instead of returning i itself.

Algorithm 16 Action: RouteLookup(i, j)

Require: $receive_i(\langle \text{“Lookup”}, user, k\rangle) \land\ status_i \neq \text{“dead”}$

1: **procedure**
2:     **if** $status_i = \text{“ready”} \land covers(lset_i, j)$ **then**
3:         $deliver(i, user, j)$
4:     **else if** $\neg covers(lset_i, j)$ **then**
5:         **if** findNext$(i, j) \neq i$ **then**
6:             send $\langle\langle \text{“Lookup”}, user, j\rangle$,findNext$(i, j)\rangle$
7:         **else**
8:             reportError $\langle \text{“NoLegalRoute”}, i, j\rangle$
9:         **end if**
10:    **end if**
11: **end procedure**

The reason for introducing this error message is to detect erroneous implementations of findNext(i, j) at a earlier stage, because the idea is to leave the freedom of implementing findNext(i, j), so that another implementation could be adopted with a simplified routing table structure. In fact, simplified versions of the routing table and findNext(i, j) were indeed used in the early stages of model checking Pastry, in order to prune the unnecessary state space and make it feasible to find bugs in the model and essential counterexamples to the protocol design.

Besides, it is not specified in action RouteLookup(i, j) here in Algorithm 16 what a node should do if it is not “ready” but covers the key j. In this case, it does not fulfill the prerequisites of the action RouteLookup(i, j) and therefore should do nothing. This means that the node will either wait until its neighborhood changes and at the same time its coverage of the keys, so that it does not cover the key anymore and will then route the message to another node or it should first complete its join process and then deliver the “Lookup” message when it is “ready”.

Algorithm 17 demonstrates an implementation of the auxiliary function *findNext* (i, j). Another simplified implementation is adopted for model checking purpose to prune the unnecessary state space caused by the routing table structure.

In Algorithm 17, node i first tries to find the closest node to the key j as the next hop from its leaf sets, when the key j lies within the range covered by its leaf set nodes, or when its leaf sets overlap (lines 2 and 3). Otherwise, it will try to find the next hop from the corresponding position in its routing table (lines 5 and 6), where the value at this position, if not *null*, should share one more digit $(r + 1)$ of its prefix than the key shares with the node. If no such value exists, it will try to combine all known node identifiers from its leaf sets and routing table and choose the closest one which shares the same length r or a longer prefix with the key j (lines 8 and 9). The result is either a different node closer to the key j, or node i itself, if no such node could be found. Note that if the leaf sets are not empty, the returned value can never be i itself.

Algorithm 17 Auxiliary function: findNext(i, j)

1: **function**
2:     **if** $\left(\begin{array}{l} overlaps(lset_i) \lor \\ (CwDist(lset_i.leftmost, j) \\ \quad \leq CwDist(lset_i.leftmost, lset_i.rightmost)) \end{array} \right)$ **then**
3:         $n :=$ pick $k \in lset_i$ such that $|k - j|$ is minimal
4:     **else**
5:         $r := \textit{shared-prefix-length}(j, i)$
6:         $n := rtable_i(r, \textit{r-th-digit}(j))$
7:         **if** $n \neq null$ **then**
8:             return n
9:         **else**
10:             $can := \{k \in lset_i \cup rtable_i : |j - k| < |j - i|$
11:                 $\land \textit{shared-prefix-length}(j, k) \geq r\}$
12:             **if** $can \neq \{\}$ **then**
13:                 $n :=$ pick $k \in can$ such that $|k - j|$ is minimal
14:                 return n
15:             **else**
16:                 return i
17:             **end if**
18:         **end if**
19:     **end if**
20: **end function**

Join

Only "ready" nodes can answer a "JoinRequest" message and one "ready" node can only let one node join at any time (see Assumptions 3.4). This restriction is made to prevent the problem illustrated in Section 3.2.4. To ensure this restriction, the variable toj_i is introduced to denote the joining node that is currently being handled by i (see Algorithm 18).

The precondition of action RecJReq(i) states that a "ready" node i only answers a "JoinRequest" message from another node j when it covers the node identifier and it is not letting any other node join the network. Unlike the action RecJReq(i) in Algorithm 6 in CastroPastry, the joining node j is added into the leaf sets of node i in order to solve the problem illustrated in Section 3.2.4. At this moment, node j is closer to node i than i's neighbor due to the properties of coverage calculation.

Algorithm 18 Action: RecJReq(i)

Require: $\begin{pmatrix} receive_i(\langle\text{"JoinRequest"}, rtable, j\rangle) \\ \wedge\ (status_i = \text{"ready"}) \wedge covers(lset_i, j) \wedge (toj_i = i) \end{pmatrix}$

1: **procedure**
2: $\quad toj_i := j$
3: $\quad lset_i := lset_i.add(\{j\})$
4: $\quad$ send $\langle\langle\text{"JoinReply"}, rtable, lset_i\rangle, j\rangle$
5: **end procedure**

When node i does not cover j, then it executes action *RouteJReq*. It tries to find the next node closer to j and forward the "JoinReply" message to it. When no such next hop can be found, it reports an error.

Algorithm 19 Action: RouteJReq(i)

Require: $\begin{pmatrix} receive_i(\langle\text{"JoinRequest"}, rtable, j\rangle) \\ \wedge\ (status_i \neq \text{"dead"}) \wedge \neg covers(lset_i, j) \end{pmatrix}$

1: **procedure**
2: $\quad$ **if** findNext(i, j) $\neq i$ **then**
3: $\quad\quad$ send $\langle\langle\text{"JoinRequest"}, rtable.add(rtable_i), j\rangle$,findNext$(i, j)\rangle$
4: $\quad$ **else**
5: $\quad\quad$ reportError $\langle\text{"NoLegalRoute"}, i, j\rangle$
6: $\quad$ **end if**
7: **end procedure**

In the event that node i covers the joining node j but has already started helping another node to join or in case node i is not yet "ready", then the "JoinRequest" message to node i will simply be delayed and handling postponed until these conditions are fulfilled.

The action RecJReply(i) in LuPastry is the same as the one in CastroPastry described in Algorithm 8, except that it adds one more precondition to the action, that

only "waiting" nodes receive "JoinReply" messages. Hence the pseudocode is omitted here. The reason for this requirement is that the arrival of a "JoinReply" message at a "dead" node obviously has no effect. Besides, after it becomes "ok" or "ready", it should not expect any "JoinReply" message, because it has already joined the network.

Probe

Unlike the action RecProbe(i) illustrated in Algorithm 10, a node can handle "Probe" messages in action RecProbe(i) only when it is active (i.e. not "dead") in the improved version illustrated in Algorithm 20, because the alternative makes no sense. Furthermore, if the node receiving a "Probe" message is not yet "ready", then it must have non-empty leaf sets. Conversely, if its leaf sets are empty, the node in question should be "ready".

This condition ensures that a "waiting" node j which has just sent its "JoinRequest" will not react to any "Probe" messages from any node n neighboring a potential replier i to its "JoinRequest" message. This is because message delay might cause j to receive the probing message from node n before i's expected "JoinReply" message. This might disturb the desired order of execution of the actions.

Another difference with Algorithm 10 in CastroPastry is that the reply to the sender contains the previous leaf sets of the probed node. Observe that in CastroPastry, the node returns the updated leaf sets, which might cause key coverage inconsistency (see Section 3.2.1).

In Algorithm 20, it is emphasized by means of the function *clone*() that the *lsetprime* is a freshly cloned, distinct object from $lset_i$, so that changes in the value of the local variable *lsetprime* will never affect the state variable $lset_i$.

Algorithm 20 Action: RecProbe(i)

Require: $\begin{pmatrix} receive_i(\langle\text{“Probe”}, j, lset, failed\rangle) \\ \wedge\ status_i \neq \text{“dead”} \\ \wedge\ (status_i = \text{“ready”} \vee \neg lset_i.isEmpty()) \end{pmatrix}$

1: **procedure**
2: $failed_i := failed_i \setminus \{j\}$
3: $rtable_i := rtable_i.add(\{j\})$
4: send $\langle\langle\text{“ProbeReply”}, i, lset_i, failed_i\rangle, j\rangle$ ▷ $lset_i$ is not changed yet
5: $lset_i := lset_i.add(\{j\})$
6: $lsetprime := lset_i.clone()$ ▷ $lset_i$ will not be modified any more.
7: $lsetprime := lsetprime.remove(failed)$
8: $lsetprime := lsetprime.add(lset \setminus failed_i)$
9: **for all** $n \in ((lset_i \cap failed) \cup (lsetprime \setminus lset_i)) \setminus (probing_i \cup failed_i)$ **do**
10: probe(i, n)
11: **end for**
12: **end procedure**

Unlike the precondition to action RecProbe(i) described in Algorithm 20, a node executing RecPRply(i) in Algorithm 21 should expect a "ProbeReply" message after

performing the probe, therefore none of the other cases described in action RecProbe(i) could occur here in the action RecPRply(i). The procedure of action RecPRply(i) (lines 3 to 11 in Algorithm 21) is similar to the action RecProbe(i) (lines 3 to 12 in Algorithm 20), except that it does not send the "ProbeReply". In contrast to Castro-Pastry Algorithm 11 illustrated in Section 3.2.4, Algorithm 21 basically only checks if the probing process is complete. In this case, a "waiting" node becomes "ok" and starts the lease granting protocol by sending "LeaseRequest" messages to its direct left and right neighbors, to confirm that its direct neighbors are either "ok" or "ready".

Algorithm 21 Action: RecPRply(i)

Require: $receive_i(\langle\text{“ProbeReply”}, j, lset, failed\rangle) \wedge status_i \neq \text{“dead”}$

1: **procedure**
2: $failed_i := failed_i \setminus \{j\}$
3: $rtable_i := rtable_i.add(\{j\})$
4: $lset_i := lset_i.add(\{j\})$
5: $lsetprime := lset_i.clone()$
6: $lsetprime := lsetprime.remove(failed)$
7: $lsetprime := lsetprime.add(lset \setminus failed_i)$
8: **for all** $n \in ((lset_i \cap failed) \cup (lsetprime \setminus lset_i)) \setminus (probing_i \cup failed_i)$ **do**
9: probe(i, n)
10: **end for**
11: $shouldBeOK := (status_i = \text{“waiting”}) \wedge (probing_i = \{\})$
12: **if** $shouldBeOK$ **then**
13: $status_i := \text{“ok”}$
14: $failed_i := \{\}$
15: send $\langle\langle\text{“LeaseRequest”}, i\rangle, lset_i.leftneighbor\rangle$
16: send $\langle\langle\text{“LeaseRequest”}, i\rangle, lset_i.rightneighbor\rangle$
17: **end if**
18: **end procedure**

Lease

The lease granting protocol modeled in LuPastry is inspired by the idea of Haeberlen et al. (2005) discussed in Section 3.2. In the action RecPRply(i) (Algorithm 21), messages of the type "LeaseRequest" are required to settle conflicts in key-coverage, we call it "lease". This part of the action is different from the periodic lease granting protocol in Haeberlen et al. (2005) and FreePastry (2009).

The periodic lease granting protocol reflects the version implemented in FreePastry (2009). In that protocol, a "ready" node periodically checks the leases of its immediate neighbors and sends "LeaseRequest" messages to those neighbors missing their leases.

The action RequestLease(i) in LuPastry Algorithm 22 describes another mechanism. The purpose of this action here is different from that in FreePastry (2009). Here it helps the joining node to complete the lease granting protocol after the first

"LeaseRequest" messages have failed to get a reply due to newly joined nodes.

Algorithm 22 Action: *RequestLease* (i)

Require: $(status_i = \text{"ok"}) \wedge (lset_i.leftneighbor \notin lease_i \vee lset_i.rightneighbor \notin lease_i)$

1: **procedure**
2: **if** $lset_i.leftneighbor \notin lease_i$ **then**
3: send $\langle\langle\text{"LeaseRequest"}, i\rangle, lset_i.leftneighbor\rangle$
4: **else if** $lset_i.rightneighbor \notin lease_i$ **then**
5: send $\langle\langle\text{"LeaseRequest"}, i\rangle, lset_i.rightneighbor\rangle$
6: **end if**
7: **end procedure**

In the following, the variables $lease_i$ and $grant_i$ are used to remember from which nodes node i received leases, and to which nodes node i granted leases, respectively.

In action RecLReq(i) (see Algorithm 23), the "LeaseReply" message is involved to answer the "LeaseRequest" message. In action RecLReply(i) (see Algorithm 24), the "LeaseReply" messages are used to propagate the information that the node has become "ready". In both cases, the node "grants" a lease. Nodes only grant leases to their direct neighbors and only "ok" or "ready" nodes can receive "LeaseRequest" or "LeaseReply" messages as a precondition to actions RecLReq(i) and RecLReply(i).

When receiving a "LeaseRequest" message in action RecLReq(i) in Algorithm 23, the node grants the leases (granting bit is set to TRUE) if the "LeaseRequest" message came from direct neighbors, otherwise it replies without granting. It was not clear in Haeberlen et al. (2005) and FreePastry (2009) if an "ok" node should be able to grant. It was discovered through model checking analysis that if only "ready" nodes can grant, then concurrent joining of nodes between "ready" nodes may never finish the join protocol to become "ready", because one of the nodes might be waiting for the other concurrent joining node to grant it. Therefore, in HAEBERLENPASTRY and LUPASTRY, both "ok" and "ready" nodes can grant.

Algorithm 23 Action: RecLReq(i)

Require: $\begin{pmatrix} receive_i(\langle\text{"LeaseRequest"}, j\rangle) \\ \wedge\ (status_i = \text{"ok"} \vee status_i = \text{"ready"}) \end{pmatrix}$

1: **procedure**
2: $isNeighbor := (j = lset_i.leftneighbor) \vee (j = lset_i.rightneighbor)$
3: **if** $isNeighbor$ **then**
4: $grant_i := grant_i \cup \{j\}$
5: **end if**
6: send $\langle\langle\text{"LeaseReply"}, lset_i, isNeighbor\rangle, j\rangle$
7: **end procedure**

In action RecLReply(i) in Algorithm 24, the "ready" or "ok" node only receives "LeaseReply" messages from its direct neighbors, because a "LeaseReply" message from another node cannot fulfill its purpose according to Haeberlen et al. (2005), as described

in Section 3.2.2: a node should transmit key coverage to its neighbor via the lease granting protocol as the last phase of its joining. When receiving a "LeaseReply" message from direct neighbors in action RecLReply(i), node j extends its $lease_j$ with the sender. If the extended $lease_j$ contains the direct left and right neighbors of node j and node j is currently still "ok", then it should become "ready".

Unlike HaeberlenPastry, one more notification step is added after a node becomes "ready", as illustrated in action RecLReply(i) in Algorithm 24. In this case, node j propagates the lease to its left and right neighbors by sending the granting message "LeaseReply" and adds these two neighbors into its grant history $grant_j$.

When the corresponding neighbor, say the left neighbor ln, receives the "LeaseReply" message from the current node j, which it has helped to join, the neighbor resets its toj_{ln} back to itself. This ensures that the "ready" node ln, which originally helped j to join, can help other nodes (see Assumption 3.4.4) only after the joining node j is "ready". Now a node has finished the join process completely with the last notifications to its neighbors.

Algorithm 24 Action: *RecLReply* (i)

Require: $\begin{pmatrix} receive_i(\langle \text{“LeaseReply”}, ls, grant\rangle) \\ \wedge\ (status_i = \text{“ok”} \vee status_i = \text{“ready”}) \\ \wedge\ (ls.node = lset_i.leftneighbor \vee ls.node = lset_i.rightneighbor) \end{pmatrix}$

1: **procedure**
2: $\quad ln := lset_i.leftneighbor$
3: $\quad rn := lset_i.rightneighbor$
4: $\quad$ **if** $grant = \text{TRUE}$ **then**
5: $\quad\quad lease_i := lease_i \cup \{ls.node\}$
6: $\quad$ **end if**
7: $\quad okToReady := (ln \in lease_i) \wedge (rn \in lease_i) \wedge (status_i = \text{“ok”})$
8: $\quad$ **if** $toj_i = ls.node$ **then**
9: $\quad\quad toj_i = i$
10: $\quad$ **end if**
11: $\quad$ **if** $okToReady$ **then**
12: $\quad\quad status_i := \text{“ready”}$
13: $\quad\quad grant_i := grant_i \cup \{ln, rn\}$
14: $\quad\quad$ send $\langle\langle \text{“LeaseReply”}, i, \text{TRUE}\rangle, ln\rangle$
15: $\quad\quad$ send $\langle\langle \text{“LeaseReply”}, i, \text{TRUE}\rangle, rn\rangle$
16: $\quad$ **end if**
17: **end procedure**

3.3.4 Verification of LuPastry

The overall verification goal is the safety property that at any time there can only be one node responsible for any key (property *CorrectDelivery*, formally defined as Property 4.3.4 in Section 4.3). The model checker TLC is employed to validate the Pastry

model illustrated in Section 3.3.3 against this property (see Section 5) for four particular nodes, in order to gain confidence, and then a theorem proving approach was embarked upon to finally verify the Pastry model for its property *CorrectDelivery* for arbitrary instances.

Reduction

First of all, the property *CorrectDelivery* is reduced to hypothetical invariants of the underlying data structures, and TLAPS is used to prove that these imply the global correctness property *CorrectDelivery*.

This property *CorrectDelivery* consists of two major parts: one states that if a node delivers a key, it should be the closest "ready" node; the other states that if this node delivers that key, then no other node should deliver it at the same time.

Knowing that only "ready" nodes can deliver and a node delivers a key when it covers it, it is intuitive to infer that the property *CorrectDelivery* is ensured if all the "ready" nodes agree with each other about their coverages. Furthermore, the coverage of a node is calculated according to the distance from a node to its direct neighbors as introduced in Section 3.1.1. Therefore, the property *CorrectDelivery* is reduced to a property of "ready" nodes and their direct neighbors: if for any "ready" node i, no other "ready" node lies between i and its immediate neighbor, then it follows that there will be no conflicts among "ready" nodes w.r.t. key coverage. This is the basic idea of the invariant *NeighborClosest*.

NeighborClosest Invariant

More precisely, the invariant *NeighborClosest* states that the left and right neighbors of any "ready" node i lie closer to i than any other "ready" node j distinct from i.

The way to prove this invariant is to focus on the critical transition when a node i goes from "ok" to "ready" in action *RecLReply* in Algorithm 24.

- This node i should ensure that there is no other "ready" node k between its direct left and right neighbors and itself.
- The node i, which is "ok" and is to become "ready", should not lie between some node r and the direct left or right neighbor of that node r.

Since node i can become "ready" only when it receives *leases* from both of its direct neighbors, there should be invariants making sure that the granted leases are indeed from the closest "ready" neighbors; in addition, the granting nodes should also ensure that there is no other "ready" node lying between themselves and the granted node.

For the first case, it is sufficient to prove a stronger invariant saying that there is no "ready" node between any (not necessarily "ready") node and its neighbors. This is in fact the invariant *IRN* illustrated in Section 6.5.1 and proved in Section 6.5.2. In addition, the basic idea of Pastry protocol that nodes always join through their closest "ready" nodes is in fact not trivial to prove, and is also formulated as the invariant *TojClosest* in Section 6.5.1.

For the second case, it is obvious that at the time of granting a lease, the nodes granted leases are neighbors of the nodes granting them (by the granting conditions in Algorithm 23), as long as no nodes joined in between and became "ready" in the meantime. In order to ensure this, further invariants are needed for the variable *grant* (e.g. *GrantNeighborNew* illustrated in Section 6.5.1). Besides, not only "ready" nodes but also "ok" nodes can grant leases, because otherwise concurrently joining nodes between two "ready" nodes might never be able to join. But if "ok" nodes are allowed to grant a lease, the induction hypothesis of *NeighborClosest* cannot be applied to prove the invariant *GrantNeighborNew*. Here again the stronger invariant *IRN*, which in fact subsumes the property *NeighborClosest*, must be applied instead of *NeighborClosest*.

Detailed proof of property *NeighborClosest* and relevant important invariants will be explained in Chapter 6.

3.4 Assumptions

Although departure of nodes itself might not hurt the property *CorrectDelivery*, it is no longer considered as a target for verification, because it does not make much sense to allow departure of nodes in the absence of any repair mechanism. The same principle applies to message loss. There is no mechanism described in Pastry as to how the lost message should be discovered and recovered. Therefore, although message loss is modeled in TLA$^+$ codes, it is not included among the possible actions of the verified Pastry model.

The following are all the assumptions for verifying the Pastry system.

Assumption 3.4.1. *There is always at least one "ready" node in the network.*

Assumption 3.4.1 makes sure that the network protocol is available, and that lookup and join services are functional. This is weaker than in Castro et al. (2004) and Haeberlen et al. (2005), both require at least L "ready" nodes.

Assumption 3.4.2. *No message manipulation or corruption occurs.*

Assumption 3.4.2 excludes security properties, which lie outside our verification interest for the moment.

Assumption 3.4.3. *No nodes leave the network.*

Assumption 3.4.3 is a very strong assumption, but at the moment no more relaxation can be found, due to the phenomenon of network separation illustrated in Section 3.2.4

Based on this assumption, the original assumption of Haeberlen et al. (2005), stating that all leaf set members are connected can be relaxed to the assumption that the members of the leaf sets of a node are always reachable from that node. This relaxed assumption makes sure that probing/routing messages can always be delivered smoothly to their destination, if it is not faulty ("dead").

Assumption 3.4.4. *At any time point, one "ready" node serves at most one joining node.*

Assumption 3.4.4 eliminates the concurrent joining problem shown in Section 3.2.4. Here *joining node* refers to a candidate node wishing/attempting to join the network which has already received a "JoinReply" message from a "ready" node in the network

Assumption 3.4.5. *No message corruption and no message loss.*

Assumption 3.4.5 allows delays such that the order in which messages are received may differ from that in which they were sent.

3.5 Summary

This chapter introduced the CastroPastry based on Castro et al. (2004), its extension HaeberlenPastry based on Haeberlen et al. (2005) and the improved Pastry IdealPastry and finally LuPastry with its formal verification.

The Pastry models CastroPastry and HaeberlenPastry are formally specified and analyzed against the formally defined correctness property of the system: *CorrectDelivery*. Non-trivial problems with these two versions are discovered by model checking analysis as illustrated in Section 3.2.4. It is revealed that leaf set completeness checks are imprecise and unnecessary. First updating and then exchanging the updated leaf sets through "ProbeReply" messages causes loss of information, and leads to violation of *CorrectDelivery*. Allowing a newly joined node to process a "Probe" message before the expected "JoinReply" message causes misunderstanding of the delayed "Probe" message in some previous join actions, which leads to violation of *CorrectDelivery*. Allowing departure of nodes with leaf set repair mechanisms enables separation of the network and violation of *CorrectDelivery*. Unrestricted concurrent joining of nodes through the same "ready" node allows new nodes to become "ready" between the original "ready" node and its neighbor, which violates *CorrectDelivery*.

The protocol is improved to IdealPastry and further to LuPastry based on the results of this analysis. The model LuPastry can be implemented according to the pseudocode demonstrated in Section 3.3.3. It is validated with model checking on 5 instances without any violation of its correctness property. Except the actions for repairing leaf sets and handling node departures from the network, it is also verified via theorem proving by postulating *CorrectDelivery* to synthesizing hypothetical invariants of the underlying data structures and then proving those invariants for networks of arbitrary size (Section 3.3.4).

This is the first attempt at verifying a real world network protocol formally and in detail. Although no solutions to the problem of guaranteeing the safety property could be found for the protocol with passive node departures and its repair mechanism, the verified protocol LuPastry providing dependable routing in spite of concurrent joining of new nodes can be also interesting for network design.

4 Detailed Formal Model of Pastry

Pastry is modeled as a (potentially infinite-state) transition system in TLA$^+$(Lamport (2002)). According to the comparison of different formal method techniques in Section 2.2, TLA$^+$ fits protocol verification quite nicely, because its concept of actions matches the rule/message-based definition of protocols.

Different versions of TLA$^+$ models of Pastry are available on the Web (see VeriDis (2013)) and this chapter illustrates LuPastry in TLA$^+$ code.

Notations from TLA$^+$ will be used based on those introduced in Section 2.3, but slightly polished for the sake of pretty representation. Several abbreviations of results of mathematical operations are used in TLA$^+$ in order to make it feasible to be proved automatically, but these are explicitly shown in the thesis. For example, 2^{M-1} is abbreviated as *half* in TLA$^+$. The invariants and theorems are always assigned names, in order to be referred to in a proof in TLA$^+$, but the assignment is omitted in the thesis to avoid confusion with the definition of data structure. The complete original TLA$^+$ model of LuPastry can be found on the Web in VeriDis (2013).

LuPastry is introduced in Section 3.3, model checked in Chapter 5 and a complete proof of correct delivery of key with all relevant interesting invariants are shown in Section 6.5.

4.1 Static Model

In the static model of Pastry, all the data structures and operations on them are modeled using the primitives provided by TLA$^+$ as introduced in Section 1.2. A data structure is always a boolean value, a natural number, a set, a function or a complex composition of them. An operation on data structure is always a functional mapping from given signature of data structures to a returned value, which is again a data structure.

4.1.1 Ring Operation

As introduced in Section 3.1.1, several parameters define the size of the ring and of the fundamental data structures. In particular, $M \in \mathbb{N}$ defines the space $I = [0..2^M - 1]$ of node and key identifiers, and $L \in \mathbb{N}$ indicates the size of each leaf set (see Definition 4.1.2). The following definition introduces different notions of distance between two nodes x and y on the ring, meaning how many identifiers lie within the region from node x to y on the ring (inclusive x and exclusive y). For example, the distance from 0 to 1 is 1 in the counter clockwise direction and $2^M - 1$ in the clockwise direction.

Definition 4.1.1 (Distances)**.** *Given x, $y \in I$:*

$$
\begin{aligned}
\mathit{Dist}(x,y) &\triangleq \text{IF } x-y < -2^{M-1} \text{ THEN } x-y+(2^M-1)\\
&\ \text{ELSE IF } x-y > 2^{M-1} \text{ THEN } x-y-(2^M-1)\\
&\ \text{ELSE } x-y\\
\mathit{AbsDist}(x,y) &\triangleq |\mathit{Dist}(x,y)|\\
\mathit{CwDist}(x,y) &\triangleq \text{IF } \mathit{Dist}(x,y) \geq 0 \text{ THEN } \mathit{Dist}(x,y)\\
&\ \text{ELSE } 2^M + \mathit{Dist}(x,y)
\end{aligned}
$$

The sign of $\mathit{Dist}(x,y)$ is positive if there are fewer identifiers on the counter-clockwise path from x to y than on the clockwise path; it is negative otherwise. The case distinction of $x-y < -2^{M-1}$ and $x-y > 2^{M-1}$ ensures that the result ranges within $[-2^{M-1}, 2^{M-1}]$, such that its absolute value is never greater than half of the capacity of the ring. The absolute value $\mathit{AbsDist}(x,y)$ gives the length of the shortest path along the ring from x to y. Finally, the clockwise distance $\mathit{CwDist}(x,y)$ returns the length of the clockwise path from x to y.

4.1.2 Leafset

The leaf set data structure *ls* of a node is modeled as a record with three components: *ls.node*, *ls.left* and *ls.right*. The first component contains the identifier of the node maintaining the leaf sets, the other two components are the two leaf sets to either side of the node. The following operations access or manipulate leaf sets.

Definition 4.1.2 (Operations on leaf sets ($ls \in LSet$, $delta \in \text{SUBSET } I$)).

$$
\begin{aligned}
LSet &\triangleq [node \in I, left \in \text{SUBSET } I, right \in \text{SUBSET } I] \\
GetLSetContent(ls) &\triangleq ls.left \cup ls.right \cup \{ls.node\} \\
EmptyLS(i) &\triangleq [node \mapsto i, left \mapsto \{\}, right \mapsto \{\}] \\
LeftNeighbor(ls) &\triangleq \text{IF } ls.left = \{\} \\
&\quad\; \text{THEN } ls.node \\
&\quad\; \text{ELSE CHOOSE } n \in ls.left : \forall p \in ls.left : \\
&\qquad\qquad CwDist(p, ls.node) \geq CwDist(n, ls.node) \\
RightNeighbor(ls) &\triangleq \text{IF } ls.right = \{\} \\
&\quad\; \text{THEN } ls.node \\
&\quad\; \text{ELSE CHOOSE } n \in ls.right : \forall q \in ls.right : \\
&\qquad\qquad CwDist(ls.node, q) \geq CwDist(ls.node, n) \\
LeftCover(ls) &\triangleq (ls.node + CwDist(LeftNeighbor(ls), ls.node) \div 2)\%2^M \\
RightCover(ls) &\triangleq (RightNeighbor(ls) + \\
&\quad\; CwDist(ls.node, RightNeighbor(ls)) \div 2 + 1)\%2^M \\
Covers(ls, k) &\triangleq CwDist(LeftCover(ls), k) \\
&\quad\; \leq CwDist(LeftCover(ls), RightCover(ls)) \\
Overlaps(ls) &\triangleq (ls.left \cap ls.right) \neq \{\} \\
AddToLSet(delta, ls) &\triangleq \text{LET } i \triangleq ls.node \\
&\qquad can \triangleq (ls.left \cup ls.right \cup delta) \setminus \{i\} \\
&\qquad newleft \triangleq \\
&\qquad\quad \text{IF } |can| \leq L \text{ THEN } can \\
&\qquad\quad \text{ELSE CHOOSE } subsetleft \in \text{SUBSET } can : \\
&\qquad\qquad\quad \wedge |subsetleft| = L \\
&\qquad\qquad\quad \wedge \forall out \in (can \setminus subsetleft), in \in subsetleft : \\
&\qquad\qquad\qquad CwDist(in, i) < CwDist(out, i) \\
&\qquad newright \triangleq \\
&\qquad\quad \text{IF } |can| \leq L \text{ THEN } can \\
&\qquad\quad \text{ELSE CHOOSE } subsetright \in \text{SUBSET } can : \\
&\qquad\qquad\quad \wedge |subsetright| = L \\
&\qquad\qquad\quad \wedge \forall out \in (can \setminus subsetright), in \in subsetright : \\
&\qquad\qquad\qquad CwDist(i, in) < CwDist(i, out) \\
&\quad\; \text{IN } [node \mapsto i, left \mapsto newleft, right \mapsto newright]
\end{aligned}
$$

The $EmptyLS(i)$ defines the empty leaf sets of a node. The $LeftNeighbor(ls)$ and $RightNeighbor(ls)$ of a node are defined as the closest nodes (also called direct neighbors) in the corresponding sides of its leaf set members. If both sides are empty, the node itself is taken as its neighbor. Therefore, a statement, that "the direct left (right) neighbor of a node is itself", is intuitively equivalent to a statement "the left (right) side of the leaf sets is empty". This property is formally specified in Lemma 4.1.3. Many such kinds of lemmas, to be found in VeriDis (2013), are formalized as primitive lemmas used for

formal proofs later.

Lemma 4.1.3 (EmptyLSNoNeighbor).

$$\begin{aligned} EmptyLSNoNeighbor \triangleq\ & \forall\ ls \in LSet : \\ & \quad \wedge\ LeftNeighbor(ls) = ls.node \\ & \quad \wedge\ RightNeighbor(ls) = ls.node \\ & \Leftrightarrow ls = EmptyLS(ls.node) \end{aligned}$$

Note that the parameter *ls* is an arbitrary instance of *LSet*. Intuitively one would expect to use a node identifier, e.g. i, as given parameter, which then will access the leaf sets of particular node i. Since the operation should access only data structures but not state variables, all the given parameters are instances of *LSet*, instead of node identifiers. This detail of parameter is crucial for the theorem proving process, because properties of local data structure like the Lemma 4.1.3 can be proved by their definition, but the properties of state variables are system property, which can be only proved inductively over all actions.

The functions $LeftCover(ls)$ and $RightCover(ls)$ of a given node define the upper bound (exclusive the edge) and lower bound (inclusive the edge) of the key coverage of its corresponding node ($ls.node$) w.r.t. its leaf sets. Recall that 2^M is the ring capacity of all identifiers. Function $Covers(ls, k)$ checks if a node k lies within the region starting from the $RightCover(ls)$ counter clockwise to the $LeftCover(ls)$ of a node ($ls.node$).

The boolean function $Overlaps(ls)$ checks if the left and right leaf sets are overlapping.

The operation $AddToLSet(delta, ls)$ adds the set of nodes *delta* into left and right sides of the leaf sets *ls*. More precisely, both left and right sides of the leaf sets in the resulting data structure ls' contain the L nodes closest to $ls.node$ among those contained in *ls* and the nodes in *delta*, according to the clockwise or counter-clockwise distance.

The formal specification of the operation $RemoveFromLSet(delta, ls)$ is omitted here because it is roughly the same as $AddToLSet(delta, ls)$: instead of adding new nodes, it removes *delta* from the whole leaf set and then lets the left and right sides of the leaf sets be rebuilt out of the rest of the members, details on the code can be found in VeriDis (2013).

4.1.3 Routing Table

Despite the fact that the structure of a routing table is shown to be irrelevant to verify the safety property *CorrectDelivery* later in Chapter 6, the position of the matrix plays an essential role in efficient routing and is also involved in finding the next hop in the routing algorithm illustrated in Algorithm 17 in Section 3.3.3. For this reason, a detailed formal description of the routing table is illustrated in Definition 4.1.4 as followed.

The routing table of a Pastry node is a mapping matrix from the positions (Pos) of the matrix to the value, which are either a node identifier or a place holder ($null$). As already introduced in Section 3.1.1, there are $M \div B$ rows and 2^B columns of the table, where M is formally defined in Section 4.1.1 and $B \in \mathbb{N}$ defines the base (2^B) of the

digits representing the node identifier. Therefore, $M \div B$ is also the number of digits of the node identifiers. Take the node identifiers illustrated in Figure 3.1 in Section 3.1.1 as an example. All the identifiers consist of $8 \div 2 = 4$ digits, which also explains that the routing table shown in Table 3.2 has 4 rows.

The *InitRTable* maps all the position of the matrix to the value *null*, which represents in fact a pure empty routing table. Note that this is not the real initial routing table for a new Pastry node, because each node should have itself in its routing table.

The function $\mathit{GetRTableContent}(rt)$ returns all the node identifiers in the routing table rt as a set. The function $\mathit{GetDig}(i, j)$ takes the row number i (starting from 1) and a node identifier j as arguments and returns the digit at the i-th position of the identifier j, e.g. GetDig(1, 1032) returns 1 and GetDig(2, 1032) returns 0. The function $\mathit{SharedDig}(i, j)$ takes two node identifiers i and j as arguments and returns the length of the shared prefix of them. The function $\mathit{GetPos}(i, j)$ calculates the corresponding position ($\langle \mathit{row}, \mathit{column} \rangle$) of the node identifier j in the routing table of a particular node i. For example as shown in Table 3.2, the corresponding position of node 3131 in the routing table of node 1032 is $\mathit{GetPos}(1032, 3131) = \langle 1, 3 \rangle$, as it is filled in the first row and the third column.

The operation $\mathit{AddDelta}(\mathit{delta}, rt, i)$ adds all the node identifiers in the set *delta* into the routing table rt of a particular node i. It first calculates for the given set of nodes all their corresponding positions in that table (*posToChange*), then put the node identifiers one by one at the expected position. Note that for one node identifier, there is a deterministic position in the table, but there can be many nodes expected to be candidates for the same position. Here the operation $\mathit{TheBetterNode}(id1, id2)$ chooses the "better" node according to user application, which is beyond of the interest of the thesis.

The operation $\mathit{AddSelf}(rt, i)$ adds node i itself into the given routing table rt of that node i. As mentioned before, a node should have itself in its routing table. Due to the specific structure, the node identifier i should occur in each row of the routing table rt after it is added into it. This is easy to see from the example illustrated in Table 3.2.

The operation $\mathit{AddToTable}(\mathit{delta}, rt, i)$ is the frequently used one later in the dynamic model in Section 4.2, when new entries in *delta* need to be added into a particular routing table rt of specific node i. It is implemented here in the way that the node identifier always shows up in each row of its routing table after the adding operation is accomplished each time. To achieve that, the operation first adds the given set of nodes except i into the routing table, then it adds the node itself into it.

Definition 4.1.4 (Operations on Routing Table ($rt \in \mathit{RTable}$, $\mathit{delta} \in \textsc{subset}\ I, i, j \in I$,

null as a new system Constant)).

$$
\begin{array}{rcl}
Pos & \triangleq & [1..(M \div B)] \times [0..(2^B - 1)] \\
RTable & \triangleq & [Pos \rightarrow (I \cup \{null\})] \\
InitRTable & \triangleq & [p \in Pos \mapsto null] \\
GetRTableContent(rt) & \triangleq & \{rt[pos] : pos \in Pos\} \setminus \{null\} \\
GetDig(i, j) & \triangleq & \text{LET } p \triangleq M \div B - i \\
 & & \text{IN } (j \% 2^{B(p+1)}) \div 2^{Bp} \\
SharedDig(i, j) & \triangleq & \text{LET } helpSD[k \in \mathbb{N}] \triangleq \\
 & & \qquad \text{IF } (k > M \div B) \vee (GetDig(i, k) \neq GetDig(j, k)) \\
 & & \qquad \text{THEN } k - 1 \\
 & & \qquad \text{ELSE } helpSD[k + 1] \\
 & & \text{IN } helpSD[1] \\
GetPos(i, j) & \triangleq & \text{LET } r \triangleq SharedDig(i, j) \\
 & & \qquad c \triangleq GetDig(j, r + 1) \\
 & & \text{IN } \langle r + 1, c \rangle \\
AddDelta(delta, rt, i) & \triangleq & \text{LET } posToChange \triangleq \{GetPos(i, j) : j \in delta\} \\
 & & \text{IN } [p \in Pos \mapsto \\
 & & \qquad \text{IF } p \in posToChange \\
 & & \qquad \text{THEN} \\
 & & \qquad \quad \text{LET } newNode \triangleq \text{CHOOSE } v \in delta : GetPos(i, v) = p \\
 & & \qquad \quad \text{IN } TheBetterNode(rt[p], newNode) \\
 & & \qquad \text{ELSE } rt[p]] \\
AddSelf(rt, i) & \triangleq & \text{LET } posToChange \triangleq \{\langle k, GetDig(i, k)\rangle : k \in 1..(M \div B)\} \\
 & & \text{IN } [p \in Pos \mapsto \\
 & & \qquad \text{IF } p \in posToChange \\
 & & \qquad \text{THEN } i \\
 & & \qquad \text{ELSE } rt[p]] \\
AddToTable(delta, rt, i) & \triangleq & AddSelf(AddDToTable(delta \setminus \{i\}, rt, i), i)
\end{array}
$$

4.1.4 Messages

Messages are defined as records consisting of their destinations and the message content: $DMsg \triangleq [destination \in I, mreq \in MReq]$. The message content($MReq$) consists of the following different types.

Definition 4.1.5. *Message Types*

$$
\begin{aligned}
Look &\triangleq [type \in \{\text{"Lookup"}\}, node \in I] \\
NoLR &\triangleq [type \in \{\text{"NoLegalRoute"}\}, key \in I] \\
JReq &\triangleq [type \in \{\text{"JoinRequest"}\}, rtable \in RTable, node \in I] \\
JRpl &\triangleq [type \in \{\text{"JoinReply"}\}, rtable \in RTable, lset \in LSet] \\
Prb &\triangleq [type \in \{\text{"Probe"}\}, node \in I, lset \in LSet, failed \in \text{SUBSET } I)] \\
PRpl &\triangleq [type \in \{\text{"ProbeReply"}\}, node \in I, lset \in LSet, failed \in \text{SUBSET } I)] \\
LReq &\triangleq [type \in \{\text{"LeaseRequest"}\}, node \in I] \\
LReply &\triangleq [type \in \{\text{"LeaseReply"}\}, lset \in LSet, grant \in \{\text{TRUE}, \text{FALSE}\}] \\
MReq &\triangleq Look \cup NoLR \cup JReq \cup JRpl \cup Prb \cup PRpl \cup LReq \cup LReply
\end{aligned}
$$

In the later formal specification, the values of the components of a message is often be referred by component name. For example, $m.mreq.type =$ "Lookup" tells that the message $m \in DMsg$ has the message content of type *Look*.

The "Lookup" message contains only the node which it is looking for.

The "NoLegalRoute" message is used for reporting an error during routing "Lookup" or "JoinRequest" messages in action RouteLookup(i, j) and RouteJReq(i, j).

In the "JoinRequest" message, *rtable* is used to collect new nodes during routing process, so that they can be added into the final receiver's routing table and replied in the "JoinReply" message to the joining node in action RecJReq(i). Together with the routing table, the leaf sets of the final receiver of "JoinRequest" message is also replied in the "JoinReply" message, to help the joining node to initiate its own ones.

The "Probe" message contains the node sending the probe (*node*), its leaf set and the set of nodes it believes to be faulty. As a reply to the probing message, the "ProbeReply" message contains the node replying the probe (*node*), the replier's leaf sets and *failed* set of nodes.

The "LeaseRequest" message contains the node identifier which request a lease. The "LeaseReply" message replies such requests, either granting the lease or not, with its component *grant*. Besides, the receiver provides also its own leaf sets, where its own identifier is contained in $ls.node$. Instead of only sending back the node identifier, the leaf sets were designed to provide extra nodes information, which may serve a purpose as it is in "Probe" message, namely to propagate and exchange the leaf sets among nodes.

4.2 Dynamic Model

The high-level outline of the transition model is formally specified in TLA$^+$ as shown Definition 4.2.1. The overall system specification *Spec* is defined as $Init \land \Box[Next]_{vars}$, which is the standard form of TLA$^+$ system specifications as introduced in Section 2.3. It requires that all runs start with a state that satisfies the initial condition *Init*, and that every transition either does not change *vars* (defined as the tuple of all state variables) or corresponds to a system transition as defined by *Next*. This form of system

specification is sufficient for proving safety properties. Since liveness properties are beyond the verification interest of this thesis, no fairness hypotheses are asserted, claiming that certain actions eventually occur.

Definition 4.2.1 (Overall Structure of the TLA$^+$ Specification of Pastry.)**.**

$$
\begin{array}{lcl}
vars & \triangleq & \langle receivedMsgs, status, lset, probing, failed, rtable, lease, grant, toj \rangle \\
Init & \triangleq & \land\ receivedMsgs = \{\} \\
 & & \land\ status = [i \in I \mapsto \text{IF } i \in A \text{ THEN } \text{"ready"} \text{ ELSE } \text{"dead"}] \\
 & & \land\ toj = [i \in I \mapsto i] \\
 & & \land\ probing = [i \in I \mapsto \{\}] \\
 & & \land\ failed = [i \in I \mapsto \{\}] \\
 & & \land\ lease = [i \in I \mapsto \text{IF } i \in A \text{ THEN } A \text{ ELSE } \{i\}] \\
 & & \land\ grant = [i \in I \mapsto \text{IF } i \in A \text{ THEN } A \text{ ELSE } \{i\}] \\
 & & \land\ lset = [i \in I \mapsto \text{IF } i \in A \\
 & & \qquad\qquad\qquad \text{THEN } AddToLSet(A, EmptyLS(i)) \\
 & & \qquad\qquad\qquad \text{ELSE } EmptyLS(i)] \\
 & & \land\ rtable = [i \in I \mapsto \text{IF } i \in A \\
 & & \qquad\qquad\qquad \text{THEN } AddToTable(A, InitRTable, i) \\
 & & \qquad\qquad\qquad \text{ELSE } AddToTable(\{i\}, InitRTable, i)] \\
Next & \triangleq & \exists i, j \in I : \lor\ Join(i,j) \lor Lookup(i,j) \lor Deliver(i,j) \\
 & & \qquad\qquad \lor\ RouteJReq(i,j) \lor RouteLookup(i,j) \\
 & & \qquad\qquad \lor\ RecJReq(i) \lor RecJReply(j) \\
 & & \qquad\qquad \lor\ RecProbe(i) \lor RecPRply(j) \\
 & & \qquad\qquad \lor\ RecLReq(i) \lor RecLReply(i) \\
 & & \qquad\qquad \lor\ RequestLease(i) \\
Spec & \triangleq & Init \land \Box[Next]_{vars}
\end{array}
$$

The variable *receivedMsgs* holds the set of messages in transit. It is assumed in the formal model that messages are never modified when they are on the way to their destination, that is, no message is corrupted.

The other variables hold arrays that assign to every node $i \in I$ its status, leaf sets, routing table, the set of nodes it is currently probing, the set of nodes it has determined to have dropped off the ring (*failed*), the node to which it has sent a join reply and not yet got confirmation if it has become "ready" (*toj*), the nodes from which it has already got the leases (*lease*) and the nodes to which it has granted its leases (*grant*).

The predicate *Init* is defined as a conjunction that initializes all variables. In particular, the model takes a parameter A indicating the set of nodes that are initially "ready".

The next-state relation *Next* is a disjunction of all possible system actions, for all pairs of identifiers $i, j \in I$. Each action is defined as a TLA$^+$ action formula. As an example, Definition 4.2.2 shows how Deliver(i,k) is formally modeled as an action in TLA$^+$. Except Deliver(i,k), each of the other actions listed here correspond to the action with the same name introduced in Section 3.3.3.

Definition 4.2.2. *Action: Deliver(i, j)*

$$
\begin{array}{l}
\mathit{Deliver}(i, j) \triangleq \\
\quad \land\ \mathit{status}[i] = \text{"ready"} \\
\quad \land\ \exists m \in \mathit{receivedMsgs} : \land\ m.\mathit{mreq}.\mathit{type} = \text{"Lookup"} \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ m.\mathit{destination} = i \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ m.\mathit{mreq}.\mathit{node} = j \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \mathit{Covers}(\mathit{lset}[i], j) \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \mathit{receivedMsgs}' = \mathit{receivedMsgs} \setminus \{m\} \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{status}, \mathit{rtable}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{lease}, \mathit{grant}, \mathit{toj} \rangle
\end{array}
$$

Different from the action Deliver(i, user, k) introduced in Algorithm 16 in Section 3.3.3, here the action Deliver(i, j) has only two parameters i (the delivering node) and j (the key) and the original sender x is omitted. That is because it is only interesting here who delivers this key, but not the recipient. The action is executable if node i is "ready", if there exists an unhandled "Lookup" message addressed to i, and if j, the identifier of the requested key, falls within the coverage of node i (see Definition 4.1.2). Its effect is simply defined as removing the message m from the network, because for this particular action, only the execution of the action is interesting, not the answer message that it generates. Each time when receiving a message, the node will remove the message from the message pool *receivedMsgs*, so that it will not be received again. The other variables are unchanged.

As follows, the corresponding formal definitions of those actions introduced in Section 3.3 will be illustrated, which is also formally verified against system property *CorrectDelivery*. This formal model of Pastry is the final formal model proved in Section 6.5. Other version of Pastry models can be found in VeriDis (2013).

The prefix *Rec* of many action names denotes that those actions are reactive on the message it received. For example, one of the precondition of the action RecProbe(i) is that the network contains a message of type "Probe" whose destination is i.

Different from the actions described in pseudocode before, changes to a variable within an action is made only once in the formal specification as illustrated in the following sections. This is because a specification in TLA$^+$ is by default a big conjunction of all formulas, as explained in Section 2.3.

4.2.1 Lookup and Join

The action Lookup(i, j) in Algorithm 1 is modeled in the Definition 4.2.3. The node i looks up the key j by sending a "Lookup" message to itself, which will then enable the routing action RouteLookup(i, j) (Definition 4.2.6) later.

Definition 4.2.3. *Action: Lookup(i, j)*

$$
\begin{array}{l}
\mathit{Lookup}(i, j) \triangleq \\
\quad \land\ \mathit{receivedMsgs}' = \mathit{receivedMsgs} \\
\qquad\qquad\qquad \cup \{[\mathit{destination} \mapsto i, \\
\qquad\qquad\qquad\qquad \mathit{mreq} \mapsto [\mathit{type} \mapsto \text{"Lookup"}, \mathit{node} \mapsto j]]\} \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{status}, \mathit{rtable}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{lease}, \mathit{grant}, \mathit{toj} \rangle
\end{array}
$$

The action Join(j, i) in Algorithm 15 is formally modeled in Definition 4.2.4, where a "dead" node j joins the network by sending a "JoinRequest" message to some node i. As described for status of nodes in Section 3.3.2, here the node changes its status from "dead" to "waiting", in order to wait for the "JoinReply" message later in action RecJReply(i).

Definition 4.2.4 (Action: Join(j, i))**.**

$$
\begin{aligned}
&\mathit{Join}(j, i) \triangleq \\
&\quad \land\ \mathit{status}[j] = \textit{"dead"} \land \mathit{status}[i] = \textit{"ready"} \\
&\quad \land\ \mathit{status}' = [\text{EXCEPT } ![j] = \textit{"waiting"}] \\
&\quad \land\ \mathit{receivedMsgs}' = \mathit{receivedMsgs} \cup \{[\mathit{destination} \mapsto i, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{mreq} \mapsto [\mathit{type} \mapsto \textit{"JoinRequest"}, \mathit{node} \mapsto j]]\} \\
&\quad \land\ \text{UNCHANGED } \langle \mathit{rtable}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{lease}, \mathit{grant}, \mathit{toj} \rangle
\end{aligned}
$$

4.2.2 Routing Actions

Two types of messages can be routed: "Lookup" messages and "JoinRequest" messages. The action RouteLookup(i, j) routes the "Lookup" messages and the action RecJReq(i, j) routes the "JoinRequest" messages. Both of them use the same routing algorithm defined in function findNext(i, j), returning a node identifier as the next hop for the node i to forward, in order to lookup the key j. An implementation of this function is already introduced in Algorithm 17 in Section 3.3.3 and here this implementation is formally specified in Definition 4.2.5.

In the formal Definition 4.2.5, those failed nodes are excluded from being the candidate of next hop because forwarding any message to them does not make much sense. Besides, it is explicitly shown what the minimal distance as verbally expressed in Algorithm 17 in Section 3.3.3 really means. Here the Functions $\mathit{LeftMost}(\mathit{lset}[i], j)$ and $\mathit{RightMost}(\mathit{lset}[i], j)$ are defined as operations for leaf sets, which returns the left most and right most node of either side of the leaf sets of a node respectively. Their formal definition is omitted because it is exactly the same as for $\mathit{LeftNeighbor}(ls, i)$ and $\mathit{RightNeighbor}(ls, i)$ shown in Definition 4.1.2 except changing the direction of the inequality.

Definition 4.2.5 (Function: findNext(i, j)).

$$
\begin{array}{l}
\mathit{findNext}(i, j) \triangleq \\
\quad \text{IF } (\mathit{Overlaps}(\mathit{lset}[i]) \lor \mathit{CwDist}(\mathit{LeftMost}(\mathit{lset}[i]), j) \leq \\
\qquad \mathit{CwDist}(\mathit{LeftMost}(\mathit{lset}[i]), \mathit{RightMost}(\mathit{lset}[i]))) \\
\quad \text{THEN LET } \mathit{lsCan} \triangleq \mathit{GetLSetContent}(\mathit{lset}[i]) \setminus \mathit{failed}[i] \\
\qquad\qquad \text{IN IF } \mathit{lsCan} \neq \{i\} \\
\qquad\qquad\quad \text{THEN LET } \mathit{next} \triangleq \text{ CHOOSE } n \in \mathit{lsCan} \setminus \{i\} : \\
\qquad\qquad\qquad\qquad\qquad\qquad \forall m \in \mathit{lsCan} : \mathit{AbsDist}(n, j) \leq \mathit{AbsDist}(m, j) \\
\qquad\qquad\qquad\quad \text{IN IF } \mathit{AbsDist}(\mathit{next}, j) = \mathit{AbsDist}(i, j) \\
\qquad\qquad\qquad\qquad \text{THEN } \mathit{next} \\
\qquad\qquad\qquad\qquad \text{ELSE CHOOSE } n \in \mathit{lsCan} : \\
\qquad\qquad\qquad\qquad\qquad \forall m \in \mathit{lsCan} : \mathit{AbsDist}(n, j) \leq \mathit{AbsDist}(m, j) \\
\qquad\qquad\quad \text{ELSE } i \\
\quad \text{ELSE LET } r \triangleq \mathit{SharedDig}(j, i) \\
\qquad\qquad\quad \mathit{rtCan} \triangleq \mathit{rtable}[i][\langle r, \mathit{GetDig}(j, r+1) \rangle] \\
\qquad\qquad \text{IN IF } \mathit{rtCan} \neq \mathit{null} \\
\qquad\qquad\quad \text{THEN } \mathit{rtCan} \\
\qquad\qquad\quad \text{ELSE} \\
\qquad\qquad\qquad \text{LET } \mathit{canrelax} \triangleq \\
\qquad\qquad\qquad\quad \{\mathit{can} \in (\mathit{GetLSetContent}(\mathit{lset}[i]) \cup \mathit{GetRTableContent}(\mathit{rtable}[i])) \setminus \mathit{failed}[i] : \\
\qquad\qquad\qquad\qquad (\mathit{AbsDist}(j, \mathit{can}) < \mathit{AbsDist}(j, i)) \land (\mathit{SharedDig}(j, \mathit{can}) \geq r)\} \\
\qquad\qquad\qquad \text{IN IF } \mathit{canrelax} \neq \{\} \\
\qquad\qquad\qquad\quad \text{THEN CHOOSE } \mathit{can} \in \mathit{canrelax} : \\
\qquad\qquad\qquad\qquad\quad \forall m \in \mathit{canrelax} : \mathit{AbsDist}(\mathit{can}, j) \leq \mathit{AbsDist}(m, j) \\
\qquad\qquad\qquad\quad \text{ELSE } i
\end{array}
$$

The action RouteLookup(i, j) described in Algorithm 16 is separated in two actions in the formal specification here: the action Deliver(i, j) delivers the "Lookup" message in Definition 4.2.2, and the action RouteLookup(i, j) routes the "Lookup" message in Definition 4.2.6. The reason for this separation is that the enabling condition of Deliver(i, j) is used to formulate the system correctness property *CorrectDelivery*, which will be formally defined in Property 4.3.4 in Section 4.3.

Definition 4.2.6 (Action: RouteLookup(i, j)).

$$
\begin{array}{l}
RouteLookup(i, j) \triangleq \\
\quad \land\ status[i] \neq \text{“dead”} \\
\quad \land\ \exists m \in receivedMsgs : \\
\qquad \land\ m.destination = i \\
\qquad \land\ m.mreq.type = \text{“Lookup”} \\
\qquad \land\ m.mreq.node = j \\
\qquad \land\ \neg Covers(lset[i], j) \\
\qquad \land\ \text{LET } nh \triangleq findNext(i, j) \\
\qquad\quad \text{IN } receivedMsgs' = (receivedMsgs \setminus \{m\}) \cup \\
\qquad\qquad \text{IF } nh \neq i \\
\qquad\qquad \text{THEN } \{[destination \mapsto nh, mreq \mapsto m.mreq]\} \\
\qquad\qquad \text{ELSE } \{[destination \mapsto i, mreq \mapsto [type \mapsto \text{“NoLegalRoute”}, key \mapsto j]]\} \\
\quad \land\ \text{UNCHANGED } \langle status, rtable, lset, probing, failed, lease, grant, toj \rangle
\end{array}
$$

As already discussed when the action RouteLookup(i, j) is illustrated in Algorithm 16, the node only routes a “Lookup” message when it does not cover its key. It invokes the function findNext(i, j) to find a closer node to j to forward the message. If the function findNext(i, j) and $Covers(lset[i], j)$ are correctly implemented as they are formal specified in Definition 4.2.5 and 4.1.2 respectively, then the case for reporting the error message “NoLegalRoute” is redundant. However this case is kept here for checking the correctness of the model, in order to cope with the situation when the function findNext(i, j) is implemented in another way.

Routing the “JoinRequest” message in action RouteJReq(i, j) (Definition 4.2.7) has a similar procedure as routing the “Lookup” message. The difference to the action RouteLookup(i, j) is that the routed message updates its routing table after each hop as described in Algorithm 18.

Definition 4.2.7 (Action: RouteJReq(i, j)).

$$
\begin{array}{l}
RouteJReq(i, j) \triangleq \\
\land\ status[i] \neq \text{“dead”} \\
\land\ \exists m \in receivedMsgs : \\
\quad \land\ m.destination = i \\
\quad \land\ m.mreq.type = \text{“JoinRequest”} \\
\quad \land\ m.mreq.node = j \\
\quad \land\ \neg Covers(lset[i], j) \\
\quad \land\ \text{LET } nh \triangleq findNext(i, j) \\
\qquad \text{IN } receivedMsgs' = (receivedMsgs \setminus \{m\}) \cup \\
\qquad\quad \text{IF } nh \neq i \\
\qquad\quad \text{THEN } \{[destination \mapsto nh, mreq \mapsto [type \mapsto \text{“JoinRequest”}, rtable \mapsto \\
\qquad\qquad\qquad AddToTable(GetRTableContent(rtable[i]), m.mreq.rtable, i), node \mapsto j]]\} \\
\qquad\quad \text{ELSE } \{[destination \mapsto i, mreq \mapsto [type \mapsto \text{“NoLegalRoute”}, key \mapsto j]]\} \\
\land\ \text{UNCHANGED } \langle status, rtable, lset, probing, failed, lease, grant, toj \rangle
\end{array}
$$

4.2.3 Actions for Join Protocol

As illustrated in Algorithm 18, a "ready" node replies a "JoinRequest" message in action RecJReq(i), when it covers that node and has not yet started helping another node to join. The Definition 4.2.8 illustrates its formal specification.

Definition 4.2.8 (Action: RecJReq(i))**.**

$$
\begin{array}{l}
\mathit{RecJReq}(i) \triangleq \\
\quad \land\ \mathit{status}[i] = \text{“}\mathit{ready}\text{”} \\
\quad \land\ \mathit{toj}[i] = i \\
\quad \land\ \exists m \in \mathit{receivedMsgs} : \\
\qquad \land\ m.\mathit{mreq}.\mathit{type} = \text{“}\mathit{JoinRequest}\text{”} \\
\qquad \land\ m.\mathit{destination} = i \\
\qquad \land\ \mathit{Covers}(\mathit{lset}[i], m.\mathit{mreq}.\mathit{node}) \\
\qquad \land\ \mathit{toj}' = [\text{EXCEPT } ![i] = m.\mathit{mreq}.\mathit{node}] \\
\qquad \land\ \mathit{lset}' = [\text{EXCEPT } ![i] = \mathit{AddToLSet}(\{m.\mathit{mreq}.\mathit{node}\}, \mathit{lset}[i])] \\
\qquad \land\ \mathit{receivedMsgs}' = (\mathit{receivedMsgs} \setminus \{m\}) \\
\qquad\qquad \cup \{[\mathit{destination} \mapsto m.\mathit{mreq}.\mathit{node}, [\mathit{type} \mapsto \text{“}\mathit{JoinReply}\text{”}, \\
\qquad\qquad\quad \mathit{rtable} \mapsto m.\mathit{mreq}.\mathit{rtable}, \mathit{lset} \mapsto \mathit{lset}[i]]]\} \\
\quad \land\ \text{UNCHANGED } \langle \mathit{status}, \mathit{rtable}, \mathit{probing}, \mathit{failed}, \mathit{lease}, \mathit{grant} \rangle
\end{array}
$$

As explained in Section 3.3.3, only "waiting" node handles "JoinReply" message. As illustrated in Algorithm 8, the node built up its own leaf sets and routing table from the ones in the received "JoinReply" message and probes the new candidates in its leaf sets to confirm their liveness. The probing process is specified formally as a new set of messages instead of a loop of probing procedure.

Definition 4.2.9 (Action: RecJReply(i))**.**

$$
\begin{array}{l}
\mathit{RecJReply}(i) \triangleq \\
\land\ \mathit{status}[i] = \text{“}\mathit{waiting}\text{”} \\
\land\ \exists m \in \mathit{receivedMsgs} : \\
\quad \land\ m.\mathit{mreq}.\mathit{type} = \text{“}\mathit{JoinReply}\text{”} \\
\quad \land\ m.\mathit{destination} = i \\
\quad \land\ \text{LET } \mathit{newlset} \triangleq \mathit{AddToLSet}(\mathit{GetLSetContent}(m.\mathit{mreq}.\mathit{lset}), \mathit{lset}[i]) \\
\qquad\quad \mathit{toprob} \triangleq \mathit{GetLSetContent}(\mathit{newlset}) \setminus \{i\} \\
\qquad \text{IN } \land\ \mathit{rtable}' = [\text{EXCEPT } ![i] = \mathit{AddToTable}(\mathit{GetLSetContent}(m.\mathit{mreq}.\mathit{lset}) \\
\qquad\qquad\qquad\qquad \cup\ \mathit{GetRTableContent}(m.\mathit{mreq}.\mathit{rtable}), \mathit{rtable}[i], i)] \\
\qquad\quad \land\ \mathit{lset}' = [\text{EXCEPT } ![i] = \mathit{newlset}] \\
\qquad\quad \land\ \mathit{probing}' = [\text{EXCEPT } ![i] = \mathit{toprob}] \\
\qquad\quad \land\ \mathit{receivedMsgs}' = (\mathit{receivedMsgs} \setminus \{m\}) \cup \mathit{ProbeSet}(i, \mathit{newlset}, \{\}, \mathit{toprob}) \\
\land\ \text{UNCHANGED } \langle \mathit{status}, \mathit{lease}, \mathit{grant}, \mathit{toj}, \mathit{failed} \rangle
\end{array}
$$

Instead of invoking a procedure as the action described in Algorithm 8, the function ProbeSet($i, ls, f, toprob$) is invoked to prepare the set of probing messages as followed in Definition 4.2.10.

Definition 4.2.10 (Function ProbeSet($i, ls, f, toprob$)).

$$
\begin{aligned}
& \mathit{ProbeSet}(i, \mathit{ls}, f, \mathit{toprob}) \triangleq \\
& \quad \{[\mathit{destination} \mapsto j, \mathit{mreq} \mapsto [\mathit{type} \mapsto \text{"Probe"}, \\
& \quad \mathit{node} \mapsto i, \mathit{lset} \mapsto \mathit{ls}, \mathit{failed} \mapsto f]] : j \in \mathit{toprob}\}
\end{aligned}
$$

Function ProbeSet($i, ls, f, toprob$) returns a set of one probing message with the given parameter: i is a node identifier, ls is leaf sets of a node, f and $toprob$ are two sets of nodes. Different from the **for** loop in Algorithm 8, Algorithm 20 and Algorithm 21, where the procedure probe(j, k) (Algorithm 9), is invoked to change the set $probing_i$ and send "Probe" messages one after another, all the changes to a variable are summarized here in formal description at one position and only one unique change of a state variable is made in an action to keep it as a valid formula.

As explained in Algorithm 20, the precondition of action RecProbe(i) in Definition 4.2.11 rules out the case when a node is waiting for the "JoinReply" message but receives a "Probe" message.

Different from the action RecProbe(i) described in Algorithm 20, in the formal specification here in Definition 4.2.11, the probing messages with different purpose are explicitly defined with different names, to make the codes more readable. The set $prb1$ stands for the leaf set members that the sender j believes to have failed and node i will then check them itself. The set $prb2$ stands for potential new leaf set members that are not yet probed after adding new nodes from the received leaf sets into the copied local ones. The set prb summarizes the final nodes that node i should probe.

Definition 4.2.11 (Action: RecProbe(i)).

$$
\begin{array}{l}
\mathit{RecProbe}(i) \triangleq \\
\quad \land\ \mathit{status}[i] \neq \text{“dead”} \\
\quad \land\ \mathit{status}[i] = \text{“ready”} \lor \mathit{lset}[i] \neq \mathit{EmptyLS}(i) \\
\quad \land\ \exists m \in \mathit{receivedMsgs} : \\
\qquad \land\ m.\mathit{mreq}.\mathit{type} = \text{“Probe”} \\
\qquad \land\ m.\mathit{destination} = i \\
\qquad \land\ \text{LET } j \triangleq m.\mathit{mreq}.\mathit{node} \\
\qquad\qquad\quad \mathit{fi} \triangleq \mathit{failed}[i] \setminus \{j\} \\
\qquad\qquad\quad \mathit{ls}1 \triangleq \mathit{AddToLSet}(\{j\}, \mathit{lset}[i]) \\
\qquad\qquad\quad \mathit{lprim} \triangleq \mathit{GetLSetContent}(\mathit{AddToLSet}(\\
\qquad\qquad\qquad\qquad (\mathit{GetLSetContent}(m.\mathit{mreq}.\mathit{lset}) \setminus \mathit{fi}), \mathit{ls}1)) \\
\qquad\qquad\quad \mathit{prb}1 \triangleq \mathit{GetLSetContent}(\mathit{ls}1) \cap m.\mathit{mreq}.\mathit{failed} \\
\qquad\qquad\quad \mathit{prb}2 \triangleq \mathit{lprim} \setminus \mathit{GetLSetContent}(\mathit{ls}1) \\
\qquad\qquad\quad \mathit{prb} \triangleq (\mathit{prb}1 \cup \mathit{prb}2) \setminus (\mathit{probing}[i] \cup \mathit{fi}) \\
\qquad\qquad\quad \mathit{newm} \triangleq [\mathit{type} \mapsto \text{“ProbeReply”}, \mathit{node} \mapsto i, \mathit{lset} \mapsto \mathit{lset}[i], \mathit{failed} \mapsto \mathit{fi}] \\
\qquad\quad \text{IN } \land\ \mathit{failed}' = [\text{EXCEPT } ![i] = \mathit{fi}] \\
\qquad\qquad\ \land\ \mathit{rtable} = [\text{EXCEPT } !'[i] = \mathit{AddToTable}(\{j\}, \mathit{rtable}[i], i)] \\
\qquad\qquad\ \land\ \mathit{lset}' = [\text{EXCEPT } ![i] = \mathit{ls}1] \\
\qquad\qquad\ \land\ \mathit{probing}' = [\text{EXCEPT } ![i] = \mathit{probing}[i] \cup \mathit{prb}] \\
\qquad\qquad\ \land\ \mathit{receivedMsgs}' = (\mathit{receivedMsgs} \\
\qquad\qquad\qquad\qquad \cup \{[\mathit{destination} \mapsto j, \mathit{mreq} \mapsto \mathit{newm}]\} \\
\qquad\qquad\qquad\qquad \cup \mathit{Probe}(i, \mathit{ls}1, \mathit{fi}, \mathit{prb})) \setminus \{m\} \\
\quad \land\ \text{UNCHANGED } \langle \mathit{status}, \mathit{lease}, \mathit{grant}, \mathit{toj} \rangle
\end{array}
$$

The formal model of action RecPRply(i) in Algorithm 21 is described in Definition 4.2.12. Similar to the formal model of action RecProbe(i) in Definition 4.2.11, the different set of “Probe” messages are explicitly defined. The set $prb1$ in both actions is the same. The set $prb1 \cup prb2$ in action RecPRply(i) is in fact the same as prb in action RecProbe(i). The set $prb3$ in action RecPRply(i) stands for the overall nodes that node i will probe and those nodes that node i has probed but not yet got any reply from. If $prb3$ is empty, the current node can become “ok” if it is still of the status “waiting”.

Definition 4.2.12 (Action: RecPRply(i)).

$$
\begin{array}{l}
RecPRply(i) \triangleq \\
\;\wedge\; status[i] \neq \text{"dead"} \\
\;\wedge\; \exists m \in receivedMsgs : \\
\quad \wedge\; m.mreq.type = \text{"ProbeReply"} \\
\quad \wedge\; m.destination = i \\
\quad \wedge\; \text{LET } j \triangleq m.mreq.node \\
\qquad\quad fi \triangleq failed[i] \setminus \{j\} \\
\qquad\quad ls1 \triangleq AddToLSet(\{j\}, lset[i]) \\
\qquad\quad lprim \triangleq GetLSetContent(AddToLSet(\\
\qquad\qquad\qquad (GetLSetContent(m.mreq.lset) \setminus fi), ls1)) \\
\qquad\quad prb1 \triangleq GetLSetContent(ls1) \cap m.mreq.failed \\
\qquad\quad prb2 \triangleq lprim \setminus (GetLSetContent(ls1) \cup probing[i] \cup fi \cup prb1) \\
\qquad\quad prb3 \triangleq (probing[i] \cup prb1 \cup prb2) \setminus \{j\} \\
\qquad\quad shouldBeOK \triangleq \wedge\; status[i] = \text{"waiting"} \\
\qquad\qquad\qquad\qquad\quad\; \wedge\; prb3 = \{\} \\
\quad\;\; \text{IN } \wedge\; rtable' = [\text{EXCEPT } ![i] = AddToTable(\{j\}, rtable[i], i)] \\
\qquad\;\; \wedge\; lset' = [\text{EXCEPT } ![i] = ls1] \\
\qquad\;\; \wedge\; failed' = [\text{EXCEPT } ![i] = \text{IF } shouldBeOK \text{ THEN } \{\} \text{ ELSE } fi] \\
\qquad\;\; \wedge\; probing' = [\text{EXCEPT } ![i] = prb3] \\
\qquad\;\; \wedge\; status' = [\text{EXCEPT } ![i] = \text{IF } shouldBeOK \text{ THEN } \text{"ok"} \text{ ELSE } status[i]] \\
\qquad\;\; \wedge\; receivedMsgs' = (receivedMsgs \\
\qquad\qquad\qquad\qquad\qquad \cup\; Probe(i, ls1, fi, prb1 \cup prb2) \\
\qquad\qquad\qquad\qquad\qquad \cup\; \text{IF } shouldBeOK \\
\qquad\qquad\qquad\qquad\qquad\quad \text{THEN } \{[destination \mapsto LeftNeighbor(ls1), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad mreq \mapsto [type \mapsto \text{"LeaseRequest"}, node \mapsto i]]\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \cup \{[destination \mapsto RightNeighbor(ls1), \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad mreq \mapsto [type \mapsto \text{"LeaseRequest"}, node \mapsto i]]\} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } \{\}) \setminus \{m\} \\
\;\wedge\; \text{UNCHANGED } \langle lease, grant, toj \rangle
\end{array}
$$

4.2.4 Request and Grant Leases

The periodic lease checking of "ok" nodes in action RequestLease(i) in Algorithm 22 is modeled as non-deterministic action in the formal model as follows in Definition 4.2.13.

Definition 4.2.13 (Action: RequestLease(i)).

$$
\begin{array}{l}
\mathit{RequestLease}(i) \triangleq \\
\land\ \mathit{status}[i] = \text{“}ok\text{”} \\
\land\ \textsc{let}\ \mathit{ln} \triangleq \mathit{LeftNeighbor}(\mathit{lset}[i]) \\
\qquad\quad \mathit{rn} \triangleq \mathit{RightNeighbor}(\mathit{lset}[i]) \\
\quad \textsc{in}\ \land\ (\mathit{ln} \notin \mathit{lease}[i] \lor \mathit{rn} \notin \mathit{lease}[i]) \\
\qquad \land\ \mathit{receivedMsgs}' = (\mathit{receivedMsgs} \\
\qquad\quad \cup\ \textsc{if}\ \mathit{ln} \notin \mathit{lease}[i] \\
\qquad\qquad \textsc{then}\ \{[\mathit{destination} \mapsto \mathit{ln}, \\
\qquad\qquad\qquad \mathit{mreq} \mapsto [\mathit{type} \mapsto \text{“}LeaseRequest\text{”}, \\
\qquad\qquad\qquad \mathit{node} \mapsto i]]\} \\
\qquad\qquad \textsc{else}\ \{\}) \\
\qquad\quad \cup\ \textsc{if}\ \mathit{rn} \notin \mathit{lease}[i] \\
\qquad\qquad \textsc{then}\ \{[\mathit{destination} \mapsto \mathit{rn}, \\
\qquad\qquad\qquad \mathit{mreq} \mapsto [\mathit{type} \mapsto \text{“}LeaseRequest\text{”}, \\
\qquad\qquad\qquad \mathit{node} \mapsto i]]\} \\
\qquad\qquad \textsc{else}\ \{\}) \\
\land\ \textsc{unchanged}\ \langle \mathit{status}, \mathit{rtable}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{lease}, \mathit{toj}, \mathit{grant} \rangle
\end{array}
$$

As illustrated in action RecLReq(i) in Algorithm 23 and RecLReply(i) in Algorithm 24, only “ok” or “ready” nodes can receive “LeaseRequest” or “LeaseReply” messages. In action RecLReq(i), a node grants the lease if the sender of the “LeaseRequest” is its direct neighbor. The Definition 4.2.14 illustrates its formal description.

Definition 4.2.14 (Action: RecLReq(i)).

$$
\begin{array}{l}
\mathit{RecLReq}(i) \triangleq \\
\land\ \mathit{status}[i] \in \{\text{“}ok\text{”}, \text{“}ready\text{”}\} \\
\land\ \exists m \in \mathit{receivedMsgs} : \\
\quad \land\ m.\mathit{mreq}.\mathit{type} = \text{“}LeaseRequest\text{”} \\
\quad \land\ m.\mathit{destination} = i \\
\quad \land\ \mathit{grant}' = [\textsc{except}\ ![i] = \\
\qquad\qquad \textsc{if}\ m.\mathit{mreq}.\mathit{node} \in \{\mathit{LeftNeighbor}(\mathit{lset}[i]), \mathit{RightNeighbor}(\mathit{lset}[i])\} \\
\qquad\qquad \textsc{then}\ \mathit{grant}[i] \cup \{m.\mathit{mreq}.\mathit{node}\} \\
\qquad\qquad \textsc{else}\ \mathit{grant}[i]] \\
\quad \land\ \mathit{receivedMsgs}' = (\mathit{receivedMsgs} \setminus \{m\}) \\
\qquad\qquad \cup\ \{[\mathit{destination} \mapsto m.\mathit{mreq}.\mathit{node}, \mathit{mreq} \mapsto \\
\qquad\qquad\quad [\mathit{type} \mapsto \text{“}LeaseReply\text{”}, \mathit{lset} \mapsto \mathit{lset}[i] \\
\qquad\qquad\quad \mathit{grant} \mapsto m.\mathit{mreq}.\mathit{node} \in \{\mathit{LeftNeighbor}(\mathit{lset}[i]), \\
\qquad\qquad\qquad \mathit{RightNeighbor}(\mathit{lset}[i])\}]]\} \\
\land\ \textsc{unchanged}\ \langle \mathit{status}, \mathit{rtable}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{lease}, \mathit{toj} \rangle
\end{array}
$$

In action RecLReply(i), a node receives only “LeaseReply” from its neighbor as explained for Algorithm 24. The formal description of the action RecLReply(i) is shown

in Definition 4.2.15. As explained in Algorithm 24, a node will upgrade its status from "ok" to "ready" if it has the leases from both its direct neighbors. Together with upgrading, it sends the "LeaseReply" messages to both of its neighbors to announce its new status. As stated by Algorithm 24, the action RecLReply(i) serves two purpose, for the joining node to receive grants and upgrade its status, and also for the "ready" node to receive grants from the joining node to release its variable *toj*, in order to accept other join requests.

Definition 4.2.15 (Action: RecLReply(i))**.**

$$
\begin{array}{l}
RecLReply(i) \triangleq \\
\;\land\; status[i] \in \{\text{"ready"}, \text{"ok"}\} \\
\;\land\; \exists m \in receivedMsgs : \\
\quad\quad \land\; m.mreq.type = \text{"LeaseReply"} \\
\quad\quad \land\; m.destination = i \\
\quad\quad \land\; m.mreq.lset.node \in \{LeftNeighbor(lset[i]), RightNeighbor(lset[i])\} \\
\quad\quad \land\; \text{LET } ln \triangleq LeftNeighbor(lset[i]) \\
\quad\quad\quad\quad\quad rn \triangleq RightNeighbor(lset[i]) \\
\quad\quad\quad\quad\quad newlease \triangleq \text{IF } m.mreq.grant = \text{TRUE} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{THEN } lease[i] \cup \{m.mreq.lset.node\} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{ELSE } lease[i] \\
\quad\quad\quad\quad\quad okToReady \triangleq \land\; ln \in newlease \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\; rn \in newlease \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\; status[i] = \text{"ok"} \\
\quad\quad\; \text{IN } \land\; lease' = [\text{EXCEPT } ![i] = newlease] \\
\quad\quad\quad\quad \land\; toj' = [\text{EXCEPT } ![i] = \text{IF } toj[i] = m.mreq.lset.node \text{ THEN } i \text{ ELSE } toj[i]] \\
\quad\quad\quad\quad \land\; status' = [\text{EXCEPT } ![i] = \text{IF } okToReady \text{ THEN } \text{"ready"} \text{ ELSE } status[i]] \\
\quad\quad\quad\quad \land\; grant' = [\text{EXCEPT } ![i] = \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{IF } okToReady \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{THEN } grant[i] \cup \{LeftNeighbor(lset[i]), RightNeighbor(lset[i])\} \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{ELSE } grant[i]] \\
\quad\quad\quad\quad \land\; receivedMsgs' = \text{IF } okToReady \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{THEN } (receivedMsgs \setminus \{m\}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \cup\; \{[destination \mapsto LeftNeighbor(lset[i]), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad mreq \mapsto [type \mapsto \text{"LeaseReply"}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad lset \mapsto lset[i], grant \mapsto \text{TRUE}]]\} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \cup\; \{[destination \mapsto RightNeighbor(lset[i]), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad mreq \mapsto [type \mapsto \text{"LeaseReply"}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad lset \mapsto lset[i], grant \mapsto \text{TRUE}]]\} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{ELSE } (receivedMsgs \setminus \{m\}) \\
\;\land\; \text{UNCHANGED } \langle lset, rtable, probing, failed \rangle
\end{array}
$$

4.2.5 Statuses

As already illustrated in Section 3.3.2, different statuses of the nodes are defined to distinguish the different phrases one node has, when it is joining the network.

Definition 4.2.16 (Status).

$$
\begin{aligned}
ReadyNodes &\triangleq \{i \in I : status[i] = \text{“ready”}\} \\
OKNodes &\triangleq \{i \in I : status[i] = \text{“ok”}\} \\
ReadyOK &\triangleq ReadyNodes \cup OKNodes \\
WaitNodes &\triangleq \{i \in I : status[i] = \text{“waiting”}\} \\
DeadNodes &\triangleq \{i \in I : status[i] = \text{“dead”}\}
\end{aligned}
$$

Here the *status* is one of the variables of the Pastry model as introduced in Definition 4.2.1. Here new names are assigned to sets of nodes with particular statuses. The status of a node i can be expressed either using variable name with indicator ($status_i =$ “ready”) or using membership of the sets defined above ($i \in$ ReadyNodes). The reason different ways are used to express the same thing is that the model checker TLC can work more efficiently by accessing directly the variables in the first way, while the second way is more readable and adequate for the theorem prover TLAPS.

It is obvious that the four subsets of nodes, as defined in Definition 4.2.16: ReadyNodes, OKNodes, WaitNodes and DeadNodes, are disjoint. As explained in Section 3.3.2, a node is initially either a member of ReadyNodes or DeadNodes. As soon as it sends the “JoinRequest” message, it becomes a member of WaitNodes, which means it is waiting to become member of OKNodes. After it has completed its leaf sets and received all the “ProbeReply” messages, it will become member of OKNodes. Once it has obtained both leases from its left and right neighbors, it will become a member of ReadyNodes. Only members of ReadyNodes can deliver “Lookup” messages or reply to “JoinRequest” messages.

4.3 The Correctness Properties

This section summarizes the correctness properties of the Pastry model that the thesis has verified.

4.3.1 Type Correctness

Since TLA$^+$ does not have type, state variables should conform to their desired data structures, such that accessing components of them will be always successful. For example, $status[i]$ should access the state variable *status* of a particular node i and it is supposed to be one of the states, not a node identifier. The correctness of “types” are defined as state property *TypeInvariant* and then proved to be an invariant for the system as shown in Theorem 4.3.1.

Theorem 4.3.1 (Type Correctness)**.** $Spec \Rightarrow \Box TypeInvariant$

The state property *TypeInvariant* is shown in Property 4.3.2, which defines an exhausted type constraints for all state variables. Most of the variables are functional mappings from node identifiers to a set of identifiers. The leaf set of a node must be an instance of data structure *LSet*. Besides, the *node* component of the leaf set must coincides with the corresponding node identifier. Since a node only allows one node to join through it at a time, the variable *toj* always maps a node identifier to another.

Property 4.3.2 (*TypeInvariant*)**.**

$$
\begin{aligned}
TypeInvariant \triangleq\ & \land\ receivedMsgs \in \textsc{subset}\ DMsg \\
& \land\ status \in [I \rightarrow \{\text{"ready"}, \text{"ok"}, \text{"waiting"}, \text{"dead"}\}] \\
& \land\ lease \in [I \rightarrow \textsc{subset}\ I] \\
& \land\ grant \in [I \rightarrow \textsc{subset}\ I] \\
& \land\ rtable \in [I \rightarrow RTable] \\
& \land\ lset \in [I \rightarrow LSet] \land\ \forall i \in I : lset[i].node = i \\
& \land\ probing \in [I \rightarrow \textsc{subset}\ I] \\
& \land\ failed \in [I \rightarrow \textsc{subset}\ I] \\
& \land\ toj \in [I \rightarrow I]
\end{aligned}
$$

4.3.2 Safety Property

The major verification goal of this thesis is to show that the Pastry protocol as formally modeled above confirms (ensures) the safety property *CorrectDelivery*, saying that the key will be delivered by the correct node, which should be the only one responsible node covering the key.

Formally speaking as Theorem 4.3.3, the verification goal is to show that given the formulas defined for Pastry as *Spec*, it can be entailed that the property *CorrectDelivery* always holds.

Theorem 4.3.3 (Correctness of Pastry)**.** $Spec \Rightarrow \Box CorrectDelivery$

As mentioned at the beginning of Chapter 3, the main safety property of Pastry is that there can be only one node responsible for any key at any time. This is formally expressed as follows.

Property 4.3.4 (*CorrectDelivery*)**.**

$$
\begin{aligned}
& CorrectDelivery \triangleq \forall i, k \in I : \\
& \quad \textsc{enabled}\ Deliver(i, k) \\
& \quad \Rightarrow\ \land\ \forall n \in I \setminus \{k\} : status[n] = \text{"ready"} \Rightarrow AbsDist(i, k) \leq AbsDist(n, k) \\
& \quad\quad\ \ \land\ \forall j \in I \setminus \{i\} : \neg\textsc{enabled}\ Deliver(j, k)
\end{aligned}
$$

The property *CorrectDelivery* asserts that whenever node i can execute the action Deliver(i, k) for key k then both of the following statements are true:

- The node i has minimal absolute distance from the key k among all the "ready" nodes in the network.
- The node i is the only node that may execute the action Deliver(i, k) for the key k.

Observe that there can be two nodes with minimal distance from k, to either side of the key. Therefore, the asymmetry in the definition of LeftCover(ls, k) and RightCover(ls, k) in Definition 4.1.2 is designed to break the tie and ensure that only one node is allowed to deliver.

From the definition of the action Deliver(i, k)(see Definition 4.2.2), it is intuitive to see that the Theorem 4.3.3 is related to the Coverage of node i w.r.t. the key k. Observing the definition of Covers(ls, k) in Definition 4.1.2, it is intuitive to tell that the correct knowledge about neighborhood of a node is the essential key to ensure correct coverage. This intuition leads to the discovery of the property *NeighborProp*.

Property 4.3.5 (*NeighborProp*)**.**

$$NeighborProp \triangleq HalfNeighbor \land NeighborClosest$$

The property *NeighborProp* consists of two properties: *HalfNeighbor* and *Neighbor-Closest*. The property *HalfNeighbor* basically asserts that whenever there is more than one "ready" or "ok" node i, then the left and right neighbors of every such node i are different from i. In other words, the leaf sets of such a node i can not be empty, followed by the lemma *EmptyLSNoNeighbor* introduced in Lemma 4.1.3 in Section 4.1.2. The other case states that some node considers itself its neighbor (on both sides) and in fact that node is the only *ReadyOK* node.

Property 4.3.6 (*HalfNeighbor*)**.**

$$\begin{aligned}
&HalfNeighbor \triangleq \\
&\quad \lor \forall k \in ReadyOK: \\
&\qquad \land RightNeighbor(lset[k]) \neq k \\
&\qquad \land LeftNeighbor(lset[k]) \neq k \\
&\quad \lor \exists k \in ReadyOK: \\
&\qquad \land ReadyOK = \{k\} \\
&\qquad \land LeftNeighbor(lset[k]) = k \\
&\qquad \land RightNeighbor(lset[k]) = k
\end{aligned}$$

The property *NeighborClosest* states that the left and right neighbors of any "ready" node i lie closer to i than any other "ready" node j.

Property 4.3.7 (*NeighborClosest*)**.**

$$\begin{aligned}
&NeighborClosest \triangleq \forall i, j \in ReadyNodes: \\
&\quad i \neq j \Rightarrow \land CwDist(LeftNeighbor(lset[i]), i) \leq CwDist(j, i) \\
&\qquad\qquad \land CwDist(i, RightNeighbor(lset[i])) \leq CwDist(i, j)
\end{aligned}$$

In the next chapter, the model checker TLC is employed to validate that the property *NeighborProp* is an invariant for the Pastry protocol specified as *spec* in Definition 4.2.1 with up to 4 nodes as instances. Finally in Chapter 6 a complete formal proof will be unfolded starting with the formal proof of reducing *CorrectDelivery* to *NeighborProp*.

5 Model Checking with TLC

This chapter explains how the model checker TLC introduced in Section 1.2 is employed to analyze and validate the Pastry models CastroPastry, HaeberlenPastry and LuPastry. In Section 5.2, CastroPastry and HaeberlenPastry are analyzed with help of the violation trace of property *CorrectDelivery* and *NeighborClosest*. Then it is demonstrated how the interesting properties of LuPastry (specified in Chapter 4), such as *Symmetry*, are analyzed. After that, Section 5.3 describes how TLC is used to validate LuPastry.

5.1 Introduction

The following screen shot in Figure 5.1 shows the user interface of the TLA$^+$ toolbox when it launches the model checker TLC on Pastry for finding and analyzing counterexamples.

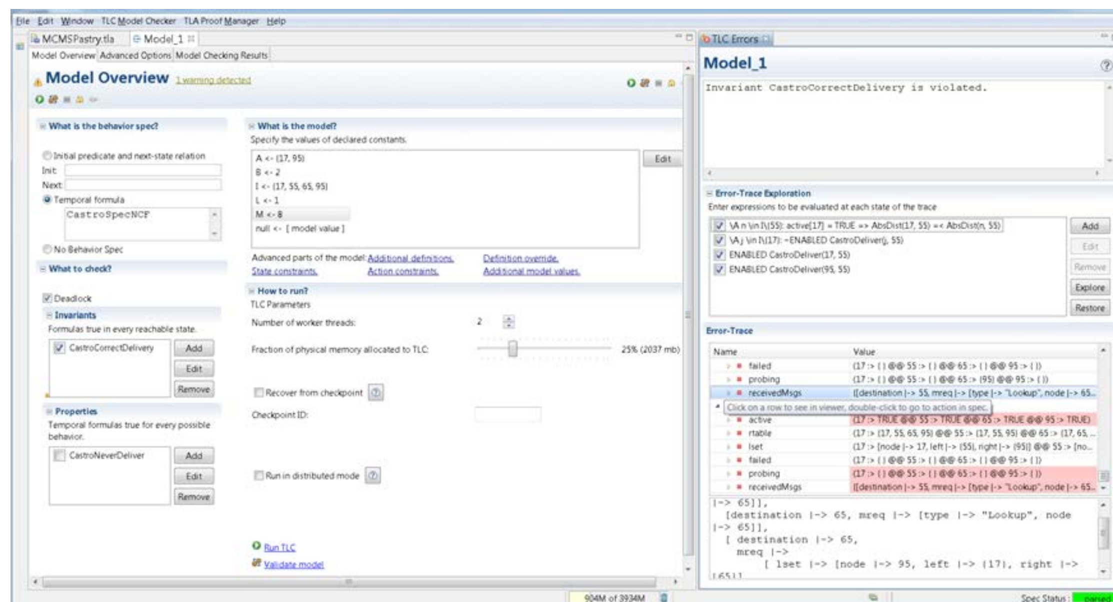

Figure 5.1: Screen shot of the TLA$^+$ Toolbox running model checker TLC.

In Figure 5.1, configuring a run of model checking is illustrated in the *Model Overview* feature in the left half of the picture. In the upper-left window, a user interface is provided to specify the initial state and the *Next* formula, or directly temporal formula

as the specification. In the windows below, the properties or invariants are added into the lists respectively.

On top of the middle in Figure 5.1, system parameters introduced in Section 3.1.1 are configured in the window. In the following sections without explicit clarification, the system parameter B as introduced in Section 3.1.1 is set to 2, meaning that the identifier of a node is represented in 4-based system as shown in the example in Figure 3.1. The parameter L for the length of leaf set as introduced in Section 3.1.1 is set to 1 and M for the maximal identifier space as introduced also in Section 3.1.1 is set to 8. In order to restrict the state space, the identifier space from $[0..2^M - 1]$ as defined in Section 4.1.1 is initialized with four arbitrary nodes $I \triangleq \{17, 55, 65, 95\}$. For convenience[1], node identifiers shown in the examples for model checking are simply written in decimal. The system parameter A is initialized with the set $\{17\}$, which means that at the beginning there is only one "ready" node in the network.

There are other options of running TLC, for example, the checkbox in the mid-left provides the option if the TLC should check deadlocks, and the counter and turning bar at the center of the screen let user configure how many threads and resources of memory should the TLC use. There are advanced options summarized in another view, where breadth-first search or depth-first search is selected. By default, TLC is run in breadth-first search. When all the configuration options are chosen, the button"Run TLC" at the bottom in the middle can be clicked to launch the model checker. Then the view of *Model Checking Results* will be shown in the front layer, demonstrating the current depth of model exploration and the number of explored states, distinct states and unexplored states in the queue.

The trace exploration features on the right half of the Figure 5.1 demonstrates the trace of the found counterexample. Changed state variables are highlighted and clicking on the line of one particular state variable unfolds the details of its value in the window below. When double clicking on the action in the trace, as the tool tips suggest, the definition predicate of this action shows up in the left half of the screen, covering the *Model Overview.* The *Error-Trace Exploration* features in the middle on the right side allows additional formulas being evaluated after a counterexample was found, in order to help find out the reason causing the violation.

This TLA$^+$ toolbox is employed to discover counterexamples for analysis purpose thanks to its interactive user interface. Before this was developed in 2011, each run of TLC had been launched by command line with help of a separate configuration file for TLC to find the assignment of all the system parameters. This traditional way is still used for generating log files for important counterexamples and for validation of the model on remote servers, where huge system resources are allocated and controlled over a long time without human attendance.

[1] According to the definition of B above, these four nodes can be interpreted as 4-based numbers for routing table and calculation of prefixes, namely $(0101)_4$, $(0313)_4$, $(1001)_4$ and $(1133)_4$. However, it is a convention in TLA$^+$ to write all the numbers as naturals in decimal base.

5.2 Model Analysis on Pastry Properties

Various models of Pastry are specified in TLA$^+$ according to different designs of Pastry and verification purposes. In this section, only the analysis on CastroPastry based on Castro et al. (2004), HaeberlenPastry based on Haeberlen et al. (2005) and LuPastry will be demonstrated. Other versions of Pastry model together with their model checking results are available online at VeriDis (2013).

Again, to restrict the state space, the routing table structure *RTable* as formalized in Definition 4.1.4 is simplified from a matrix to a set as specified in Definition 5.2.1. Correspondingly, the operations on routing tables are also simplified to set operations. For example the $AddToTable(delta, rt, i)$ for adding new entries into a routing table is simplified to a union of the two sets rt and $delta$.

Definition 5.2.1 (Simplified Routing Table ($rt \in RTable, delta \in \text{SUBSET } I, n \in I$)**.**

$$\begin{aligned} RTable &\triangleq \text{SUBSET } I \\ InitRTable &\triangleq \{\} \\ AddToTable(delta, rt, n) &\triangleq delta \cup rt \end{aligned}$$

Since the position of a node within the routing table is no more relevant to the simplified version, the function findNext(i, j) as formalized in Definition 4.2.5 is also simplified as shown in Definition 5.2.2, in the way that finding the specific entry position from the routing table is omitted. If the node is not covered within the range of leaf sets, it simply takes all nodes from its routing table and leaf sets and choose the one with minimal distance to the key.

Definition 5.2.2 (Simplified Function: findNext(i, j))**.**

$$\begin{aligned}
&findNext(i, j) \triangleq \\
&\quad \text{LET } lsCan \triangleq GetLSetContent(lset[i]) \setminus failed[i] \\
&\quad\quad\quad canrelax \triangleq \{can \in (GetLSetContent(lset[i]) \cup GetRTableContent(rtable[i])) \\
&\quad\quad\quad\quad\quad\quad \setminus failed[i] : AbsDist(j, can) < AbsDist(j, i)\} \\
&\quad \text{IN IF } Overlaps(lset[i]) \lor CwDist(LeftMost(lset[i]), j) \leq \\
&\quad\quad\quad CwDist(LeftMost(lset[i]), RightMost(lset[i])) \\
&\quad\quad \text{THEN CHOOSE } n \in lsCan : \forall m \in lsCan : AbsDist(n, j) \leq AbsDist(m, j) \\
&\quad\quad \text{ELSE IF } canrelax \neq \{\} \\
&\quad\quad\quad \text{THEN CHOOSE } can \in canrelax : \forall m \in canrelax : AbsDist(can, j) \leq AbsDist(m, j) \\
&\quad\quad\quad \text{ELSE } i
\end{aligned}$$

5.2.1 Analysis of CorrectDelivery on CastroPastry

Initial results on model checking the Pastry specification appear in Lu et al. (2010). In this paper, a problem is found for joining new nodes in CastroPastry. If two nodes join concurrently between two neighbor "ready" nodes, then the leaf sets of the new nodes may be incomplete, eventually leading to a situation where both nodes claim

Row	Action	$lset_a$	$lset_b$	$lset_c$	$lset_d$
0	*Join* (b, c) || *Join* (a, d)	N/A	N/A	{d}	{a}
1	*RecJReq* (c, b) || *RecJReq* (d, a)	N/A	N/A	{d}	{c}
2	*RecJReply* (b, c)	N/A	**{c, d}**	{d}	{c}
3	*RecJReply* (a, d)	**{c, d}**	{c, d}	{d}	{c}
4	*probe* (a, c) || ... || *probe* (b, d)	{c, d}	{c, d}	{d}	{c}
5	*RecProbe* (c, a)	{c, d}	{c, d}	{d, **a**}	{c}
6	*RecProbe* (d, b)	{c, d}	{c, d}	{d, a}	{c, **b**}
7	*RecProbe* (c, b)	{c, d}	{c, d}	{d, **b**}	{c, b}
8	*RecProbe* (d, a)	{c, d}	{c, d}	{d, b}	{c, **a**}
9	*RecPRply* (a, c) || ... || *RecPRply* (b, d)	{c, d}	{c, d}	{d, b}	{c, a}
10	*Lookup* (a, b) || *Lookup* (b, b)	{c, d}	{c, d}	{d, b}	{c, a}

Table 5.1: Concurrent Join Violation of *CorrectDelivery* in CastroPastry

responsibility for the same key. This problem was introduced in Section 3.2.1 and here it will be explained in details how TLC model checker found the counterexample.

The model checker TLC is run in its breath-first mode and it has discovered a violation of the property *CorrectDelivery*, introduced in Property 4.3.4 when it reaches the depth of 17, after it has explored 490915 states, of which 47193 states are distinct. Figure 5.1 shows the screen shot how TLC demonstrates the violation and Figure 5.2 illustrates this violation trace in detail together with Table 5.1.

In this and later sections, violation traces of model checking specific properties will be illustrated in an abbreviated way, in the sense that not each state after applying a transition action will be illustrated, but only the "milestone" states, which show major changes of the state variable. These abbreviated states are shown with step numbers in a sequence starting from initial state 0 as shown in Figure 5.2 and Table 5.1.

Initially nodes c and d are "ready" and the other nodes a and b are "dead" as illustrated in Step 0 in Figure 5.2. Here the notation of *status* ("ready" or "dead") replaces the original notation *active* (TRUE or FALSE) used in CastroPastry to keep the notion consistent with later examples, where intermediate statuses such as "waiting" and "ok" are introduced.

Two nodes a and b concurrently join between nodes c and d as shown in Step 1 in Table 5.1. According to their location on the ring as shown in Step 0 in Figure 5.2, the join request from node b is handled by node c, the join request from node a is handled by node d. Subsequently, node b receives the "JoinReply" message from c and node a from d as shown in Step 2 and 3 respectively in Figure 5.2 and Table 5.1. Same as in Figure 3.12 illustrated in Section 3.3.1, the solid arrow from node b to c on the clockwise arc shows that node b considers c as its direct clockwise neighbor, while the counter clockwise arc from node b to d represents that node d is the direct counter-clockwise neighbor of node b.

Both nodes a and b learn about the presence of c and d, and add them to their leaf sets, then send probe requests to both c and d as shown in Step 4 in Table 5.1.

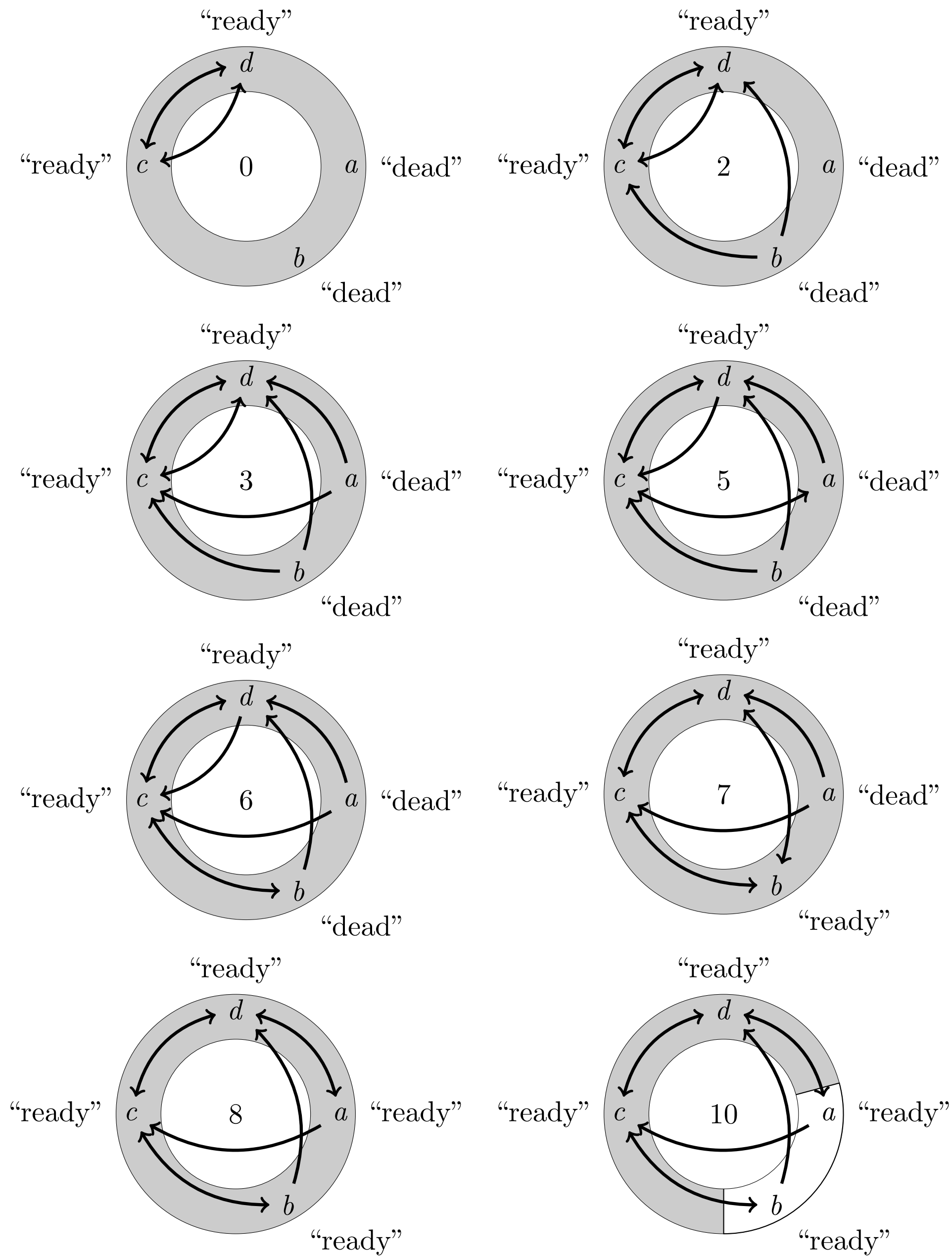

Figure 5.2: Violation trace of concurrent join in CastroPastry.

Now, node c first receives the "Probe" message from node a as show in Step 5. Learning that a new node has joined, which is closer than the previous entry in the respective leaf sets, node c updates its leaf sets with a. Subsequently node c receives the "Probe" message from node b. It learns of the existence of a new node b to its left, which is closer to the one currently in its leaf sets (node a) and therefore updates its leaf sets by replacing a with b, then sends this updated leaf sets within it "ProbeReply" message back to b as shown in Step 6.

Similarly after that, node d first handles the "Probe" message from node b as shown in Step 7 and at last it receives "Probe" message from a, similarly it replaces node b from its leaf sets with node a and replies a with this leaf sets in the "ProbeReply" message as shown in Step 8.

The subsequent transitions are abbreviated as Step 9 in Table 5.1, because receiving the "ProbeReply" messages does not increase any information of node a and b, due to the updated leaf sets in the "ProbeReply" messages as mentioned before.

The violation occurs when "Lookup" messages for the key b coming concurrently to node a and node b as Step 10 illustrated in Table 5.1. According to the shared coverage of node a and b illustrated as white sector in Step 10 in Figure 5.2, the "Lookup" message for the key b can be delivered both by a and b and hence violates the property *CorrectDelivery*.

The designers of Pastry point to a technical report Haeberlen et al. (2005) that provides a solution for this problem, according to which an updated model HAEBERLEN-PASTRY was specified, as described in Section 3.2.2 and analyzed in details in the next section.

5.2.2 Analysis of NeighborProp on HaeberlenPastry

HAEBERLENPASTRY adds the lease granting protocol after the nodes has received all the "ProbeReply" messages from candidate leaf set members. By analyzing HAEBERLEN-PASTRY as reported in **?**, TLC is unable to find counterexample when it has been stopped after running on the breath-first search mode for almost one day till the depth of 29, exploring 1,697,559,249 states (123,813,996 distinct states). However, a counterexample illustrated later will show that it does not hold, which is discovered after running TLC for about 5 days till the depth of 34, exploring 23,782,311,896 states (1,303,220,907 distinct states). The results are summarized in Table 5.3 by the end of this section.

In fact, the property *CorrectDelivery*, is reduced to the property *NeighborProp* in order to launch TLC in more efficient way to discover the potential counterexample. The reason is that TLC has to be launched with lookup and routing protocol to violate the property *CorrectDelivery*. These actions can be omitted for finding violation of *NeighborProp*, which saves a huge state space. The correctness of reduction is proved using TLAPS, which will be explained later in Section 6.1.

Since the invariant *NeighborProp* consists of two separate invariant *HalfNeighbor* and *NeighborClosest*, these two invariants are analyzed separately, in order to identify which one of them is indeed violated.

By analyzing property *NeighborClosest* on HAEBERLENPASTRY, TLC has discovered

a counter example against *NeighborClosest* introduced in Section 3.2.4. Although the analysis leads to several improvements of the model, which then makes reproducing of the counterexample impossible, analysis result based on the output gives enough details of how a counterexample may occur. Here the details are explored in Figure 5.3.

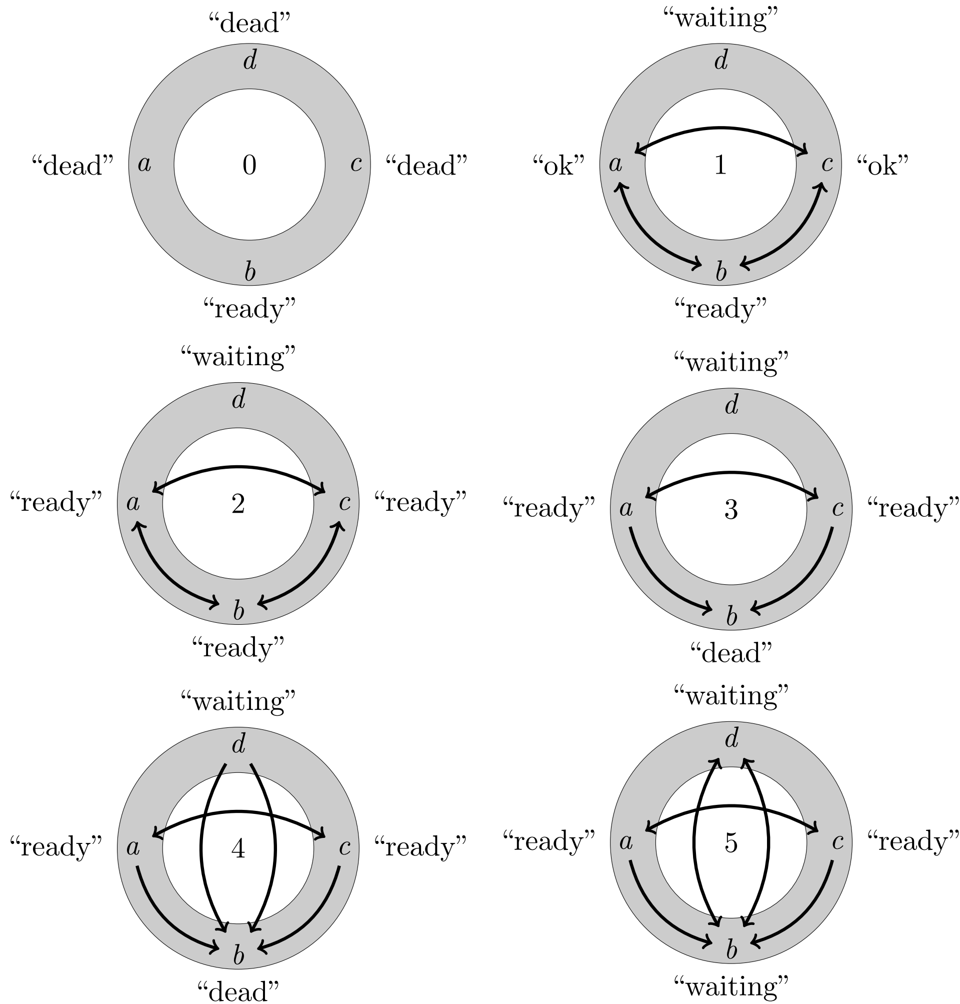

Figure 5.3: Violation trace of departure and rejoin of nodes in HaeberlenPastry (part 1).

Initially, only node b is "ready" and all the other three nodes are "dead" as illustrated in Figure 5.3. Then, nodes a, c and d concurrently join by b. Observe that there is no restriction on how many nodes can join by one node. Since b is the only "ready" node, all "JoinRequest" messages contains node b as their destination.

In the next transitions, node b replies the "JoinRequest" messages from the nodes a, c and d one after another. The "JoinReply" message sent from b to d is delayed, while nodes a and c receive the "JoinReply" messages from b and probes b back. They

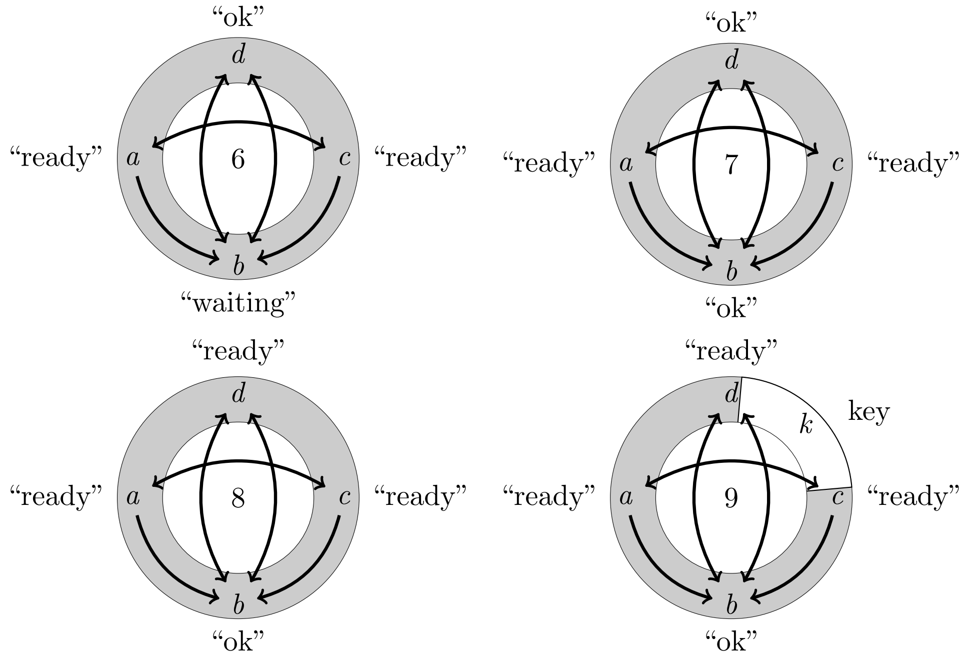

Figure 5.4: Violation trace of departure and rejoin of nodes in HaeberlenPastry (part 2).

also probe each other, such that they finally become "ok" as illustrated in Step 1 in Figure 5.3.

Subsequently, nodes a and c request leases from their neighbors $\{c, b\}$ and $\{b, a\}$, then node b replies the "LeaseRequest" messages to a and c, as well as nodes a and c reply to each other's "LeaseRequest" messages. Hence, nodes a and c become "ready" as shown in Step 2 illustrated in Figure 5.3.

Eventually node b leaves the network silently as Step 3 illustrated in Figure 5.3. By this time, the delayed message from the previously "ready" node b to the waiting node d is received by node d, which makes node d believe that b were the only "ready" node on the network. So node d probes b in Step 4 illustrated in Figure 5.3. Regard that nodes a and c are the real "ready" neighbor of d according to their positions on the ring, but none of them are aware of node d.

Node b then rejoined the network by sending "JoinRequest" message to node a. Then, node b processed the probing message from d and replied it. As a waiting node, b should handle "Probe" messages. As an effect, node b adds d as its only leaf set member and sends "ProbeReply" message back to node d. This change is summarized as Step 5 illustrated in Figure 5.3.

When node d received the "ProbeReply" messages from node b and has no other nodes as new candidates to probe, hence became "ok", as the Step 6 in Figure 5.4.

Remember that node b has tried to join through node a, but node a has not yet received this "JoinRequest" message from b. Since there is no mechanism to make

sure that node b must wait for the “JoinReply” message to become “ok”, any incoming “ProbeReply” message may finish the probing process of node b and make it “ok”.

Notice that this example is based on HaeberlenPastry with the node departure and mechanism for repairing the leaf sets illustrated in the flow chart in Figure 3.6. Here node b repairs its leaf sets by probing node d as it is the only node in its leaf sets. This “Probe” message will be replied by node d, which considers b as its only neighbor. Subsequently, node b receives the “ProbeReply” message from node d and hence become “ok”, as illustrated in the seventh stage in Figure 5.4.

As final step, node d requests the lease from node b, which then grants the lease because it is an “ok” node. Therefore, node d becomes “ready”, considering the node b as its only neighbor, whereas node a and c are ignored by node d, as illustrated in the eighth stage in Figure 5.4.

Consequently, the coverage between node d and node c becomes shared by these two nodes as the white sector shown in the last stage in Figure 5.4. This is a violation of the invariant *NeighborClosest*. When eventually a “Lookup” message for a key k within this range comes out, both of nodes c and d are able to deliver this message, which violates the property *CorrectDelivery*. Discovering the violation of *NeighborClosest* has already taken more than a day

This counterexample shows that allowing a joining node waiting for the “JoinReply” message to receive “Probe” message can cause disorder of the join process of a node and hence violates the *NeighborProp* property. A final fix to prevent this case is to add a precondition for handling any incoming “Probe” message , that only “ready” nodes or nodes that already have some leaf set members, can handle a “Probe” message, as illustrated in Algorithm 20 and specified in Definition 4.2.11, which ensures that they have already received the “JoinReply” message.

However, this improvement is not sufficient to ensure the safety property, because further counterexamples can occur when separation of the network occurs or many nodes join concurrently by one node, which will be discussed later in in Section 6.4.1 and Section 6.4.2.

5.2.3 Analysis of Invariants of Final Model of Pastry

After several improvements of the Pastry model with help of counter examples, TLC cannot find any more violation of the invariant *NeighborClosest* after running for more than five weeks, reaching the depth of 34. Having gained the confidence of correctness of the final Pastry model as illustrated in Section 3.3 and specified in Chapter 4, inductive proof of the property *NeighborClosest* has been started with the idea of extending the invariants to prove their conjunctions as inductive invariants. The following example illustrates how TLC was used to refute a candidate invariant during that approach.

Here the property *NeighborClosest* is reduced again to two symmetrical invariants: one saying that there exist no “ready” node between a “ready” node and its right neighbor; the other saying that there exist no “ready” node between a “ready” node and its left neighbor. Then it is interesting to see if the leaf set membership is symmetrical, because the ring of identifiers of Pastry is a symmetrical geometric object. Moreover,

the left and right leaf set structures on the ring are symmetrical.

Property 5.2.3 (*Symmetry* of leaf set membership)**.**

$$
\begin{aligned}
&Symmetry \triangleq \\
&\quad \forall i, j \in I : status[i] = \text{“ready”} \land status[j] = \text{“ready”} \\
&\qquad\qquad \Rightarrow (i \in GetLSetContent(lset[j]) \Leftrightarrow j \in GetLSetContent(lset[i]))
\end{aligned}
$$

It is intuitive that during the join process, the symmetry of leaf set membership does not hold, but what about for those nodes in a stable stage? The invariant above is talking about two “ready” nodes, which have already finished their joining process, so it is not obvious to see why this can be violated. Unfortunately, the property *Symmetry* is violated during the execution of the join protocol, where TLC yields the following counterexample as illustrated in Figure 5.5.

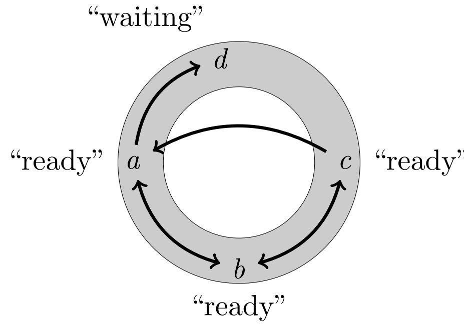

Figure 5.5: Violation trace of property *Symmetry* in LuPastry.

Suppose that three “ready” nodes a, b and c are on the ring and a node d between nodes a and c wants to join, as illustrated in Figure 5.5. The node a has received the “JoinRequest” message from d and according to the improved join protocol in action *RecJReq* illustrated in Algorithm 18 and specified in Definition 4.2.8, node a replaces node c by d in its leaf sets (assuming a leaf set size of 1). Thus, the symmetry is broken, because node c still has node a in its leaf sets.

Violation of *Symmetry* indicates that showing a “ready” node a is the direct left neighbor of another “ready” node c can not prove that node c is the direct right neighbor of the node a.

5.3 Validation

It is all too easy to introduce errors into a model that prevent the system from ever performing any useful transition, it is therefore important to ensure that nodes can successfully perform *Deliver* actions or execute the join protocol described in Section 4.2. The model checker TLC was used with assertions of their impossibility, using the following formulas.

5.3.1 Successful Join and Deliver

First of all, the Pastry model should be able to deliver a "Lookup" message as well as let a new node join.

Property 5.3.1 (*NeverDeliver* and *NeverJoin*)**.**

$$\begin{array}{lcl} \mathit{NeverDeliver} & \triangleq & \forall i, j \in I : \Box[\neg \mathit{Deliver}(i, j)]_{vars} \\ \mathit{NeverJoin} & \triangleq & \forall j \in I \setminus A : \Box(\mathit{status}[j] \neq \text{"ready"}) \end{array}$$

The first formula asserts that the *Deliver* action can never be executed, for any $i, j \in I$. Similarly, the second formula asserts that the only nodes that may ever become "ready" are those in the set A of nodes initialized to be "ready". The model checker produces counter examples to these claims within 1 second, which evidenced the possibility of successful join of node and deliver of key. This empirical result based on validation of LuPastry is shown in the first two lines in the Table 5.2 in Section 5.3.3.

5.3.2 Successful Concurrent Join of Nodes

Being confident that join and lookup services are available in the model as shown above, more advanced functionality tests are conducted in a similar way. For example, model checking formulas *ConcurrentJoin* and *CcJoinDeliver* yields counterexamples that demonstrate how two nodes may join concurrently in close proximity to the same existing node, and how they may subsequently execute *Deliver* actions for keys for which they acquired responsibility.

Property 5.3.2 (*ConcurrentJoin*)**.**

$$\begin{array}{lcl} \mathit{LightNext} & \triangleq & \exists i, j \in I : \lor\ \mathit{Deliver}(i, j) \\ & & \qquad\qquad \lor\ \mathit{RouteJReq}(i, j) \lor \mathit{RouteLookup}(i, j) \\ & & \qquad\qquad \lor\ \mathit{RecJReq}(i) \lor \mathit{RecJReply}(j) \\ & & \qquad\qquad \lor\ \mathit{RecProbe}(i) \lor \mathit{RecPRply}(j) \\ & & \qquad\qquad \lor\ \mathit{RecLReq}(i) \lor \mathit{RecLReply}(i) \\ & & \qquad\qquad \lor\ \mathit{RequestLease}(i) \lor \mathit{MsgLost} \\ \mathit{MCCJA} & \triangleq & \{17, 95\} \\ \mathit{CJNext} & \triangleq & \lor\ \mathit{Join}(65, 95) \\ & & \lor\ \mathit{Join}(55, 17) \\ & & \lor\ \mathit{Join}(70, 95) \\ & & \lor\ \mathit{LightNext} \\ \mathit{SpecConcJoin} & \triangleq & \mathit{Init} \land \Box[\mathit{CJNext}]_{vars} \\ \mathit{ConcurrentJoin} & \triangleq & \Box\neg(\mathit{status}[95] = \text{"ready"} \\ & & \quad \land\ \mathit{status}[17] = \text{"ready"} \\ & & \quad \land\ \mathit{status}[65] = \text{"ready"} \\ & & \quad \land\ \mathit{status}[70] = \text{"ready"} \\ & & \quad \land\ \mathit{status}[55] = \text{"ready"}) \end{array}$$

In these model checking cases, such as the example illustrated in Property 5.3.2, concrete examples are constructed to prevent the state explosion problem. Instead of using the *Next* formula defined in the general specification of Pastry in Definition 4.2.1, where the action *Join* and *Lookup* can be invoked for all possible nodes, restricted set of actions is defined in model checking experiments. In this example Property 5.3.2, *LightNext* is different from *Next* in the way that no *Join* or *Lookup* actions are included. Then the specific *Join* and *Lookup* actions with particular nodes are added to the set of possible actions as in *CJNext* in Property 5.3.2 to see if these nodes could be joined or if a particular key could be delivered.

In Property 5.3.2 a specific model checking scenario is constructed for the property *ConcurrentJoin* starting with two "ready" nodes: $MCCJA \triangleq \{17, 95\}$. The new parameter *MCCJA* is defined to replace the system parameter A introduced in Definition 4.2.1 for the initiating "ready" nodes. Another way to do this is to initiate the system parameter A directly by launching the model checker each time. The new specification *SpecConcJoin* is formalized to replace the specification *Spec* in Definition 4.2.1, where the restricted *LightNext* is extended with three particular joins of nodes into the network.

After running TLC for 4 seconds on the model *SpecConcJoin* in LuPastry, it finds a violation trace of the property *ConcurrentJoin* as shown in the third line of Table 5.2, illustrating how these nodes have concurrently joined into the network after exploring 27,802 states (2348 distinct states) when it reaches the depth of breath-first search by 23.

Another model checking scenario is constructed for property *CcJoinDeliver* in Property 5.3.3, starting with the system parameter: $MCCJDA \triangleq \{17\}$, meaning only one "ready" node 17 is initially in the network. The specification *SpecCcJD* is formalized with the disjunction *CCJDNext*, which extends the *LightNext* (as first introduced in Property 5.3.2) with joining action of node 95 and 55. Furthermore, the *Lookup* action is added into *CCJDNext* to lookup a key 65 by node 95.

After running TLC on LuPastry for 2 seconds, it finds a violation trace, illustrating how these nodes have concurrently joined into the network and provided a successful lookup service after exploring 18,119 states (1619 distinct states) in the depth of 23.

Property 5.3.3 (*CcJoinDeliver*)**.**

$$
\begin{array}{lll}
MCCJDA & \triangleq & \{17\} \\
CCJDNext & \triangleq & \lor\ Join(95, 17) \\
 & & \lor\ Join(55, 17) \\
 & & \lor\ Lookup(95, 65) \\
 & & \lor\ LightNext \\
SpecCcJD & \triangleq & Init \land \Box[CCJDNext]_{vars} \\
CcJoinDeliver & \triangleq & \Box[\nexists i, j \in I : Deliver(i, j) \\
 & & \quad \land\ status[95] = \text{"ready"} \\
 & & \quad \land\ status[17] = \text{"ready"} \\
 & & \quad \land\ status[55] = \text{"ready"}]_{vars}
\end{array}
$$

Properties	Time	Depth	# States	counterexample
NeverDeliver	< 1”	4	9	yes
NeverJoin	< 1”	9	28	yes
ConcurrentJoin	4”	23	27,802	yes
CcJoinDeliver	4”	23	18,119	yes
Symmetry	4”	24	26,970	yes
NeighborProp	> 5 weeks	34	331,300,918,096	no
CorrectDelivery	> 5 weeks	34	331,300,918,096	no

Table 5.2: Validation results of LuPastry with 4 nodes, leaf set length $L = 1$

5.3.3 Summary of Model Checking Results

After the Pastry model was improved based on the counterexamples illustrated in Section 5.2, the final Pastry model LuPastry, which is specified in Chapter 4, is validated against the property *CorrectDelivery* and *NeighborProp* in the previous section.

Table 5.2 summarizes the model checking experiments described so far for 4 nodes. The model checker TLC is launched in command line mode with 40 worker threads (hyper-threaded on 20 CPUs) on a 64 Bit Linux server with Xeon(R) CPU E7-4860 running at 2.27GHz with 256 GB of shared memory.

For each run, Table 5.2 reports the running time, the number of states generated until TLC found a counterexample (or, in the case of Property 4.3.4, until the process is killed), and the largest depth of these states. For example, the verification of Property 4.3.4 is stopped after running TLC for more than five weeks without any counterexample as illustrated in the last two lines in Table 5.2. Since the model checker TLC is run in breadth-first search mode, if the model contains a counterexample to this property, it must be of depth at least 34.

As introduced in the previous chapter, the *NeighborProp* property is the conjunction $\mathit{HalfNeighbor} \wedge \mathit{NeighborClosest}$, to which the safety property *CorrectDelivery* can be reduced (the proof is shown later in Section 6.1). The property *HalfNeighbor* here simply states that a node either has nodes in both sides of its leaf sets or it has empty leaf sets, which rules out the situation that a node has only half of the leaf sets.

With the strong assumption that no nodes leave the network, the validation of 3 nodes on the model with complex routing table structure as specified in Chapter 4 is completed within about an hour. The model checker TLC has found $199,466,497$ states, of which $2,293,760$ states are distinct. It reaches depth of 38. The validation of Pastry on 4 nodes is not completed within 5 weeks, as illustrated in Table 5.2.

However, on the simplified Pastry models, where routing table is totally removed and node are only allowed to join and lookup, the model checker has completed the validation of LuPastry after 2 hours 15 minutes. It has found 16,064,479 states, 504,624 distinct states and it has reached the depth of 49.

On the same simplified version of LuPastry, a validation with 5 nodes was completed after running the model checker for a week without any counterexamples. It has

Pastry Version (# Nodes)	Time	Depth	# States	# Distinct States	Result
CastroPastry (4)	30 sec	17	490,915	47,193	Violated
Haeberlen-Pastry (4)	> 20 hours	29	1,698 Mio.	124 Mio	Interrupted
Simple Haeberlen-Pastry (4)	5 days	34	23,782 Mio.	1,303 Mio.	Violated
LuPastry (3)	1 hour	38	199 Mio.	2 Mio.	Validated
LuPastry (4)	> 5 weeks	34	331,300 Mio.	1,984 Mio.	Interrupted
Simple LuPastry (4)	2 hours 15 min	49	16 Mio.	0.5 Mio.	Validated
Simple LuPastry (5)	1 week	54	7,070 Mio.	142 Mio.	Validated

Table 5.3: Model Checking Property *CorrectDelivery* for different Pastry models with different nodes, leaf set length $L = 1$

explored 7,070,872,610 states, under which 141,786,762 are distinct. It reached the depth of 54. Verifying this simplified version leads to the conclusion that for a network of 5 particular nodes, LuPastry is completely correct under the assumption that no nodes leave the network.

Table 5.3 summarizes the model checking results mentioned above verifying the property *CorrectDelivery* on different Pastry models (CastroPastry, Haeberlen-Pastry and LuPastry) and with different nodes (3, 4 and 5).

5.4 Experiences and Best Practices

The use of TLC helped to quickly explore different alternatives and ask "what-if" questions, and in this way produced different models are produced corresponding to possible implementation choices.

The trace exploration features provided by TLC through the TLA$^+$ toolbox were invaluable for understanding counterexamples produced for corner cases and for improving the models.

However, TLC suffer from state explosion for complex systems as the one in this thesis. Once TLC did no longer quickly produce error traces, one can turn to the command-line version and let the model checker run on a multi-processor server. The models shown above generate more than 30 billion states even for instances with just four nodes; hash collisions are therefore highly probable. The use of several worker threads during state exploration led to a significant speed-up and worked without a glitch.

Following are some best practices for using TLC to analyze complex system, in particular to work around state explosion problem.

First, there are several useful options provided by TLC. This thesis runs TLC using shell script by following command.

```
java m.tla -difftrace -cleanup -deadlock -workers 40 -config m.cfg
```

Here the `m.tla` and `m.cfg` are the model checking TLA$^+$ file and its corresponding configuration file. The "deacklock" is used for checking the deadlock, "workers" for running it with more threads to boots up the speed, "difftrace" for restricting the output of violation trace such that it only shows the changed state variables for each step.

Second, the way to prune the state space is to avoid symmetries as shown in Property 5.3.2. Here actions Join(i, j) and Lookup(i, j) are removed by *LightNext* and particular join actions are added to avoid the permutation of all possible joins of nodes. A risk has to be noticed here that using this approach, the TLC can be only used to find counterexamples. However, if the TLC provides no counterexamples but completely explored all states, no confidence of the model can be gained.

Third, an advanced method to prune the state space is to reduce the complexity of the data structure or even remove the "unnecessary" state variables. In this thesis, the routing table is shown to be irrelevant for guaranteeing safety property. Hence, removing routing table gains efficient discovery of potential design problems of protocol.

Fourth, it is recommended to write the TLA$^+$ and the configuration files for TLC more modular, such that different versions can be easily constructed for model checking. In this thesis, definitions relevant to the ring, leaf set, routing tables and messages are encapsulated in separate modules and the inclusion structure of the modules is like a shell. Recall that TLA$^+$ does not allow second declaration or definition of the same name. Therefore, more complex structure is not possible. In this way, the module for routing table can be removed with minimal changes of the model. Besides, a separate model checking process is launched using separate configuration file for each validation purpose. This makes it possible to have a statistical comparison with evidences as illustrated in Table 5.3.

6 Formal Proof of the Property CorrectDelivery

The theorem proving approach starts with HaeberlenPastry but it fails because a counterexample of *NeighborClosest* is discovered during the proof of HaeberlenPastry. In a first step, hypothetical invariants *HalfNeighbor* and *NeighborClosest* of the underlying data structures are stated. Note that they do not have arguments because all state variables are implicitly the arguments of all properties. *HalfNeighbor* states that the leaf set of a "ready" or "ok" node is never empty (contains only itself) except that it is the only "ready" or "ok" node on the ring. *NeighborClosest* basically states that no "ready" node lies between a "ready" node and its direct neighbor according to its leaf set. TLAPS, the interactive TLA$^+$ proof system (Chaudhuri et al. (2010)), is used to prove that these imply the global correctness property *CorrectDelivery*, which states that a key is always delivered by the closest "ready" node and no other "ready" nodes deliver it at the same time. The reduction result is published in **?** and Lu et al. (2011). The reduction proof is explained in Section 6.1.

Section 6.2 illustrates the inductive approaches for proving property *NeighborProp*, which is conjunction of *HalfNeighbor* and *NeighborClosest*. First, a detailed proof of property *HalfNeighbor* on LuPastry (which was originally constructed for an early version of LuPastry but then adapted to LuPastry) is explained to demonstrate how the invariant is extended during the proof and is finally proved inductively. Then, the problem of finding the inductive invariants for proving *NeighborClosest* on LuPastry is illustrated.

Subsequently, Section 6.3 demonstrates an inductive proof of a stronger invariant *CompleteLeafSet* on IdealPastry (an intermediate version of LuPastry), subsuming the invariant *NeighborClosest*, but postulating two strong hypotheses that no nodes are allowed to leave the network and no nodes can concurrently join between two "ready" nodes close to each other. *NeighborClosest* basically states that no "ready" nodes lies between any node with non-empty leaf sets and its direct neighbors according to its leaf sets. IdealPastry shares the same definitions for actions of HaeberlenPastry except that it has a new variable *toj* and the preconditions of RecJReq(i) checks of the neighbor's *toj* value, to implement the lock, namely, a node i does not answer a "JoinRequest" message if itself or its neighbors currently help new node to join. Besides, it improves several draw backs of HaeberlenPastry according to Section 3.2.4. It uses a passive way to unlock the *toj*: The lock will be released when the "ready" node i non-deterministically requires its leases from the joining node j after it has become "ready". By receiving a "LeaseReply" message from j, the node i releases its *toj* lock. The formal model of IdealPastry and its proof are available online at VeriDis (2013).

After that, Section 6.4 illustrates how the assumptions are relaxed. The assumption that no nodes leave the network cannot be relaxed because separation of the network may occur as shown in Section 6.4.1. Concurrent join can be allowed with a relaxed restriction that a "ready" node can handle at most one joining node at a time, due to the many concurrent join problem shown in Section 6.4.2. According to the relaxed assumption, Pastry is further modified to the final version of LuPastry, and as a consequence, many parts of the previous inductive proof of IdealPastry become invalid. The proof is then analyzed using TLAPS and new invariants are constructed to adapt the proof for the new protocol.

An inductive proof of the set of invariants on LuPastry (introduced in Section 3.3 and specified formally in Chapter 4) is illustrated in Section 6.5. The invariants are formally specified with a short intuitive introduction and an example of the proof on one core induction *IRN*, which subsumes *NeighborClosest*. *IRN* basically states that the distance between any node and its direct neighbor is less or equal to the distance from that node to any (other) "ready" nodes.

Finally, the general structure of TLA$^+$ proof codes is demonstrated as the last part of this chapter. A complete TLA$^+$ proof are available online at in VeriDis (2013).

6.1 Reducing CorrectDelivery to NeighborProp

Using TLAPS, it is proved that *NeighborProp*, introduced in Section 4.3, implies *CorrectDelivery* (Property 4.3.4). The formal proof is reduced to the proofs of the following two lemmas: *CoverageLemma* and *DisjointCovers*.

Lemma 6.1.1 shows that assuming *NeighborProp* (*NeighborClosest* and *HalfNeighbor*), for any two different "ready" nodes $i \neq n$ and key k, then if node i covers k, then i must be at least as close to k as n.

Lemma 6.1.1 (Coverage Lemma)**.**

$$\begin{aligned} & \mathit{HalfNeighbor} \land \mathit{NeighborClosest} \\ & \Rightarrow \ \forall i, n \in \mathit{ReadyNodes} : \forall k \in I : i \neq n \land \mathit{Covers}(\mathit{lset}[i], k) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathit{AbsDist}(i, k) \leq \mathit{AbsDist}(n, k) \end{aligned}$$

Proof. For the sake of a contradiction, assume that k is covered by i but that there exists another "ready" node $n \neq i$, which is closer to k than i as illustrated in Figure 6.1, then try to derive FALSE. This can occur in two cases: n and i are on the same side of k, hence n lies between i and k, or nodes i and n lie on opposite sides of the key k, but n is closer. Further distinguishing between the left and right neighborhoods of k leads to four cases, of which two are sketched as follows. The other two cases are symmetrical.

Case 1: The node n and i are both to the left of k as illustrated in Figure 6.1. According to the definition of *Cover* (Definition 4.1.2), the clockwise distance from i to its right cover (*RightCover*(*lset*[i]) in Definition 4.1.2) is strictly smaller than the clockwise distance from i to *rn* (the right neighbor of node i). As shown in the Case 1 in Figure 6.1, node *rn* is the right neighbor of node i and hence the right coverage of

node i is only a half from node i to node rn, as shown in the white sector. Here in order to let node i cover k, node rn must locate farther than twice of the distance from i to k.

Notice that if the right neighbor of node i were itself, it could cover every node on the ring. But one of the assumptions of *CoverageLemma*, the invariant *HalfNeighbor* (defined in Invariant 4.3.6), states that if the right neighbor of a "ready" node is itself, then it must be the only "ready" node on the ring. Since there are two "ready" nodes, it is ensured that $rn \neq i$.

Moreover, by the assumption at the beginning of the proof, that n is closer to k than i, and the condition for Case 1 that n and i are both to the left of k, the position of node n must be chosen between node i and k, as illustrated in the Case 1 in Figure 6.1.

Using again the definition of *Cover*, it is straightforward to tell that the clockwise distance from i to n is less than to its right cover. By transitivity, it follows that the distance from i to n is smaller than the distance from i to rn, as illustrated in the Case 1 in Figure 6.1.

Another assumption of *CoverageLemma*, the invariant *NeighborClosest* (defined as Invariant 4.3.7) states that the distance between i and its right neighbor rn is at most as large as the distance between i and any other "ready" node n. Hence a contradiction of the position of node n has been derived, which proves that in Case 1, the *CoverageLemma* holds.

Case 2: Now suppose that i is to the left and n to the right of k as illustrated in Case 2 in Figure 6.1. Since i covers k, the distance between node i and k is at most as large as the distance between i and the right cover of i, which is half the distance between i and its right neighbor rn.

According to the assumption *NeighborClosest*, the distance from i to rn is less than the distance from i to n. Hence n must be chosen in the lower half circle in the CASE 2 of Figure 6.1, such that the clockwise distance from i to n is larger than the clockwise distance from i to its right neighbor rn. This has the effect that with the above conclusion, the clockwise distance from i to k is less than half the clockwise distance from i and n. Therefore, k is closer to i than to n, which contradicts the assumption at the beginning of the proof.

□

The lemma *DisjointCovers* defined in Lemma 6.1.2 states that under the same hypotheses as *CoverageLemma*, if a "ready" node i covers key k then a different "ready" node n cannot cover k at the same time.

Lemma 6.1.2 (Disjoint Covers)**.**

$$\begin{aligned}&\mathit{HalfNeighbor} \wedge \mathit{NeighborClosest}\\ &\Rightarrow\ \forall i, n \in \mathit{ReadyNodes} : \forall k \in I : i \neq n \wedge \mathit{Covers}(\mathit{lset}[i], k)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \neg \mathit{Covers}(\mathit{lset}[n], k)\end{aligned}$$

Proof. This lemma is also proved by contradiction. Assume that both nodes i and n cover key k. By *CoverageLemma* in Lemma 6.1.1, the distances of i and n to k are the

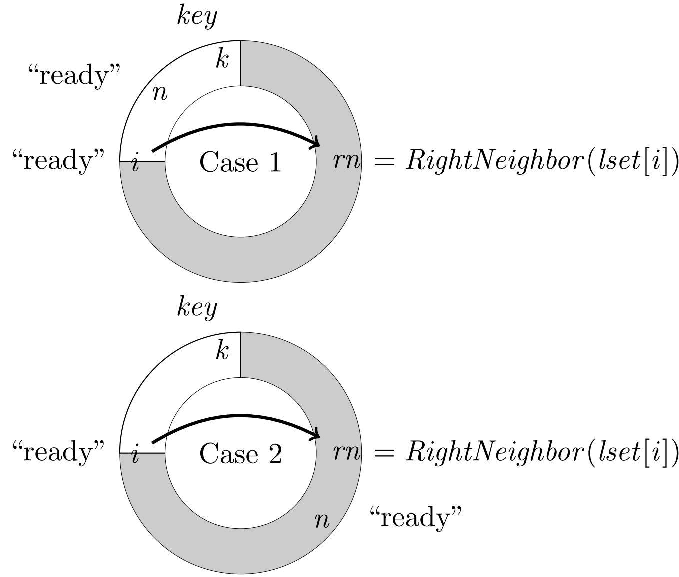

Figure 6.1: Case analysis for the proof sketch of *CoverageLemma* (Lemma 6.1.1).

same, i.e. k lies in the middle between i and n. W.l.o.g. assume that i is to the left of n as illustrated in Figure 6.2. Because i and n are not identical and both of them are "ready", *HalfNeighbor* implies that $RightNeighbor(lset[i]) \neq i$.

On one hand, by assumption *NeighborClosest*, no "ready" node lies between a "ready" node and its direct neighbor, so the distance from i to n must be at least that from i to rn. On the other hand, by the assumption that i covers k and the definition of coverage in Definition 4.1.2, the distance from i to $RightNeighbor(lset[i])$ is at least twice the distance from i to k. Satisfying both hands, the $RightNeighbor(lset[i])$ can only be node n, as illustrated in Figure 6.2.

Now n is shown to be the right neighbor of i. The asymmetrical definitions of *LeftCover* and *RightCover* in Definition 4.1.2 imply that node i does not share the coverage with its direct neighbor, which contradict the assumption at the beginning of this proof.

□

Theorem 6.1.3 (Reduction)**.**

$$HalfNeighbor \wedge NeighborClosest \Rightarrow CorrectDelivery.$$

Taking together Lemma 6.1.1 and Lemma 6.1.2, Theorem 6.1.3 follows straightforward by unfolding the definition of the property names: *HalfNeighbor*, *NeighborClosest* and *CorrectDelivery*. The formal verification in TLAPS is conducted in more complex way. First, to avoid the problem introduced by the keyword ENABLED of the action *Deliver* (cf. Definition 4.2.2). It is replaced with the enabling conditions of the action.

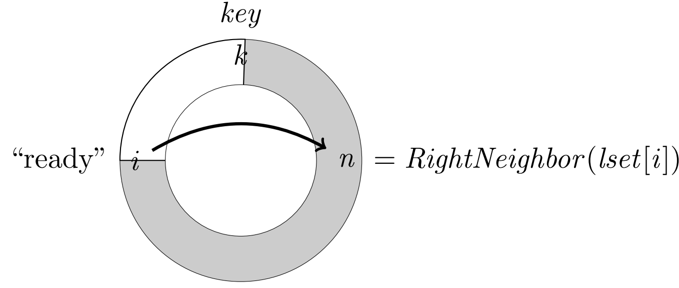

Figure 6.2: Case analysis for the proof sketch of *DisjointCovers* (Lemma 6.1.2).

The reformulated property has the following definition:

$$
\begin{array}{l}
\mathit{CorrectCoverage} \triangleq \forall i, k \in I : \\
\quad \land\ \mathit{status}[i] = \text{“ready”} \\
\quad \land\ \exists m \in \mathit{receivedMsgs} : \land\ m.\mathit{mreq.type} = \text{“Lookup”} \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ m.\mathit{destination} = i \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ m.\mathit{mreq.node} = k \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \mathit{Covers}(\mathit{lset}[i], k) \\
\quad \Rightarrow \land\ \forall n \in I : \mathit{status}[n] = \text{“ready”} \Rightarrow \mathit{AbsDist}(i, k) \leq \mathit{AbsDist}(n, k) \\
\qquad \land\ \nexists j \in I \setminus \{i\} : \land\ \mathit{status}[j] = \text{“ready”} \\
\qquad\qquad\qquad\qquad\quad \land\ \exists m \in \mathit{receivedMsgs} : \land\ m.\mathit{mreq.type} = \text{“Lookup”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ m.\mathit{destination} = j \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ m.\mathit{mreq.node} = k \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ \mathit{Covers}(\mathit{lset}[j], k)
\end{array}
$$

Then, the premises of the reduction is extended with the type correctness introduced in Property 4.3.2 and the conjunction of all inductive invariants *InvIdealPastry*, which is introduced later in Definition 6.5.2, such that the theorem is rewritten to the following form:

$$\mathit{TypeInvariant} \land \mathit{Inv} \land \mathit{HalfNeighbor} \land \mathit{NeighborClosest} \Rightarrow \mathit{CorrectCoverage}$$

This theorem is proved using Lemma 6.1.1 and Lemma 6.1.2, together with the definitions of the mentioned property names *HalfNeighbor*, *NeighborClosest* and *CorrectCoverage*. The proof is then automatically verified using TLAPS.

6.2 Towards an Inductive Proof of NeighborProp

In order to complete the proof that the specification of Pastry model (Definition 4.2.1) satisfies the property *CorrectDelivery* (Property 4.3.4), it is enough by Theorem 6.1.3 to show that every reachable state satisfies properties *HalfNeighbor* and *NeighborClosest*. A predicate *InvIdealPastry*, initially consisting of *NeighborClosest* and *HalfNeighbor* ($\mathit{InvIdealPastry} \triangleq \mathit{NeighborClosest} \land \mathit{HalfNeighbor}$) but then being strengthened, is

defined and proved as inductive invariants using the induction rule as introduced in Section 2.3.

The proof showing that the initial state satisfies *InvIdealPastry* is relative easy and hence omitted to explain. In addition, a proof for the type correctness (Theorem 4.3.1) is conducted also inductively, which is also conducted straightforward and hence omitted the discussion. This theorem is used as lemma for proving other invariants, because type correctness guarantees the access of the state variable is successful.

All the TLA$^+$ proofs can be found in VeriDis (2013). The most subtle part was the searching for inductive hypothesis for proving *HalfNeighbor*, which will be discussed as follows.

6.2.1 Inductive Proof of Invariant HalfNeighbor

The property *HalfNeighbor* in restated as follows according to Definition 4.3.6 together with *TypeInvariant* introduced in Property 4.3.2 is not sufficient to be the inductive hypothesis, in order to derive after each execution of an action, that *HalfNeighbor'* still holds. Recall that primed variables refer to the values after the action as introduced in Section 2.3.4.

$$
\begin{array}{l}
\mathit{HalfNeighbor} \triangleq \\
\quad \lor \forall k \in \text{ReadyOK} : \\
\quad\quad \land\ \mathit{RightNeighbor}(\mathit{lset}[k]) \neq k \\
\quad\quad \land\ \mathit{LeftNeighbor}(\mathit{lset}[k]) \neq k \\
\quad \lor \exists k \in \text{ReadyOK} : \\
\quad\quad \land\ \mathit{ReadyOK} = \{k\} \\
\quad\quad \land\ \mathit{LeftNeighbor}(\mathit{lset}[k]) = k \\
\quad\quad \land\ \mathit{RightNeighbor}(\mathit{lset}[k]) = k
\end{array}
$$

Adding *NeighborClosest* into the inductive hypothesis will increase the complexity of the invariant and make it even more infeasible. Hence, the *HalfNeighbor* is extended during the invariant proof, such that it is only sufficient to prove itself.

More precisely, the appropriate extensions of *HalfNeighbor* are discovered case by case when checking what is missing as prerequisites to prove *HalfNeighbor'* on its inductive proof at each action, and then strengthening *HalfNeighbor* by adding auxiliary conjunctions in such way that it provides exactly the prerequisite for the proof. Each time the invariant is extended, the model checker TLC is employed on the Pastry model to help check if the new invariant holds on the model of four nodes. By violation of such model checking approach, the formula derived from the last state of the counterexample is used to reformulate the invariant. Details are illustrated as follows.

The methodology of the proof is by contradiction, namely, try to figure out why a violation of the proof goal is not possible. Starting from the negated proving goal as post condition of each action, it is either to contradict that the post condition is not possible after applying the action, or to show that the precondition of the action contradict already with the invariant, such that it is not applicable at all.

In the proof of *HalfNeighbor*, starting from the original definition of *HalfNeighbor* in Definition 4.3.6, the proof idea is to analyze the critical transition making *HalfNeighbor'* not equal to *HalfNeighbor*. The state variables involved here are *lset* and *status*, where *lset* is explicitly shown in the formula but *status* is implicitly involved by the set *ReadyOK*, introduced in Definition 4.2.16 in Section 4.2.5.

The first task is to focus on the critical transition changing the set *ReadyOK*. Here it is only interesting to focus on the case when *ReadyOK* is increased by a node, because removing a node from the set *ReadyOK* cannot falsify *HalfNeighbor*: if the removed node is the only node in *ReadyOK*, then the set *ReadyOK'* will be empty, which automatically satisfy the first disjunct; if it is not, then by induction hypothesis, the rest of the nodes satisfy the first disjunct.

The only possibility for a node to be added into the set *ReadyOK* is when it receives a "JoinReply" message and turns to "ok" from "waiting". Hence proving that *HalfNeighbor* $\wedge$ RecPRply(i) $\Rightarrow$ *HalfNeighbor'* is the crucial part of the induction proof w.r.t. the changes on *status*. But the proof is so subtle that the original property *HalfNeighbor* has to be extended in order to prove this step. The reason is explained as follows.

According to the induction hypothesis of property *HalfNeighbor*, there are two cases to analyze as the state before execution of the action RecPRply(i): either there are more than one *ReadyOK* nodes on the ring, the leaf sets of which are not empty, or there is only one *ReadyOK* node on ring, and it has no other node in its leaf sets.

The first case is easy to prove, because in action RecPRply(i) as specified in Definition 4.2.12, node i adds the node $m.mreq.node$ into its leaf sets, which must be different from i, because no node sends "ProbeReply" to itself (which is another invariant added as extension for the proof). Therefore, the leaf sets of node i after execution of this action contains at least the newly added node, which makes it for sure not empty. Using induction hypothesis, other nodes in the set *ReadyOK* has not changed their leaf sets, therefore still maintaining the invariant *HalfNeighbor*.

The proof for the second case is a bit tricky. Since there is only one *ReadyOK* node before the execution of the action, which has no other nodes in its leaf sets, this node should not change its leaf sets according to the RecPRply(i) in Definition 4.2.12, which specifies only the $status[i]$ is changed. Therefore, after the joining node i becomes "ok" through this action, there are more nodes in the set *ReadyOK*, which should all confirm to the other case that they all have some other nodes in their leaf sets. This seems to be a contradiction leading to violation of the property *HalfNeighbor*. In fact, it is not contradiction, because the second case as predecessor state enabling the action should never occur. But exclusion of this particular case cannot be derived from the current hypothesis. Here, a new hypothesis is added:

Lemma 6.2.1 (ExclusionLemma)**.**

$$
\begin{aligned}
&\nexists a \in I, ms \in \textit{receivedMsgs} :\\
&\quad \wedge\ \textit{ReadyOK} = \{a\}\\
&\quad \wedge\ \textit{LeftNeighbor}(\textit{lset}[a]) = a\\
&\quad \wedge\ \textit{RightNeighbor}(\textit{lset}[a]) = a\\
&\quad \wedge\ ms.mreq.type = \textit{“ProbeReply”}
\end{aligned}
$$

Lemma 6.2.1 is added into the invariant *InvIdealPastry*, such that it can be used as induction hypothesis for each action. Here using Lemma 6.2.1 and the case condition, there exists no “ProbeReply” message in *receivedMsgs*, which violates the enabling condition of the action RecPRply(i) that node i receives a “ProbeReply” message. Hence, this proof branch is closed. In fact, the Lemma 6.2.1 is then merged into the extended definition of the invariant *HalfNeighbor* specified in Definition 6.2.2 as the only subformula in the conjunction talking about “ProbeReply” message.

The second task of the proof is to analyze the actions changing the variable *lset*, which are RecJReq(i), RecJReply(i), probe(i) and RecPRply(i).

The action RecPRply(i) is already considered above. The action RecJReply(i) with enabling condition of “waiting” node is no more interesting, because the precondition of the invariant is not even fulfilled, it is then valid without proof. The action probe(i) is similar to the action RecPRply(i) with respect to the change on leaf set. Therefore, its inductive proof looks similar to the one for action RecPRply(i) and thus omitted here.

Considering the action RecJReq(i), *HalfNeighbor* is preserved by the fact that after the “ready” node received the “JoinRequest” message, it will add the node into its leaf set according to the definition of RecJReq(i), which fulfills the first disjunct of the property *HalfNeighbor*.

Of course, the added sub formula in the extended invariant *HalfNeighbor* based on *LemmaExclusion* needs to be proved again inductively throughout all actions, which then causes further extension of the invariant *HalfNeighbor*. This complete invariant proof can be found on the Web in VeriDis (2013).

Finally, the property *HalfNeighbor* is extended together with its formal proof to the following complex invariant.

Invariant 6.2.2 (*HalfNeighborExt* (Extended *HalfNeighbor*))**.**

$$
\begin{array}{l}
\vee \forall k \in ReadyOK : \\
\quad \wedge\ RightNeighbor(lset[k]) \neq k \\
\quad \wedge\ LeftNeighbor(lset[k]) \neq k \\
\vee \exists k \in ReadyOK : \\
\quad \wedge\ ReadyOK = \{k\} \\
\quad \wedge\ LeftNeighbor(lset[k]) = k \\
\quad \wedge\ RightNeighbor(lset[k]) = k \\
\quad \wedge\ \forall w \in NodesWait : GetLSetContent(lset[w]) \in \text{SUBSET } \{k, w\} \\
\quad \wedge\ \neg\exists ms \in receivedMsgs : ms.mreq.type = \text{“ProbeReply”} \\
\quad \wedge\ \neg\exists mk \in receivedMsgs : \wedge\ mk.mreq.type = \text{“Probe”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ mk.destination \neq k \\
\quad \wedge\ \forall mj \in receivedMsgs : mj.mreq.type = \text{“JoinReply”} \\
\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow GetLSetContent(mj.mreq.lset) = k \\
\quad \wedge\ \neg\exists mb \in receivedMsgs : mb.mreq.type = \text{“LeaseReply”}
\end{array}
$$

The formula above merges the Lemma 6.2.1 into the original definition of *HalfNeighbor* and further extensions to make itself as an inductive invariant. *HalfNeighborExt* says that if there is more than one member of ReadyOK on the ring, then none of them will have empty leaf set.

For the special case that there is only one member of ReadyOK nodes k on the ring, following statements hold:

- it has no neighbor;
- every “waiting” node (waiting to become “ok”) knows at most the node k and itself;
- there is no “Probe” message to k;
- there is no “ProbeReply” message or “LeaseReply” message at all;
- the leaf set within “JoinReply” message can only contain k.

The idea of extending the invariant during the search of proof is also used for the invariant *NeighborClosest*, which is illustrated in the next section.

6.2.2 Towards an Inductive Proof of Invariant NeighborClosest

The inductive proof of *NeighborClosest* is far more complicated than *HalfNeighbor*, such that it is no more feasible to extend the formula in the same way, which would then generate a huge and complicated formula that would no longer be understandable. Therefore, separate understandable invariants are searched to extend *InvIdealPastry* together with *HalfNeighbor* and *NeighborClosest*, which then should mutually prove each other inductively throughout the execution of actions.

The way of searching for the appropriate invariant is backwards symbolic execution. The idea is to find a candidate invariant whose violation trace, if it is not valid, can be shorter, such that the model checker TLC can be used to help discover and improve such invariant.

The methodology starts with the violation of the property *NeighborClosest*. As illustrated in both rings in Figure 6.3. Suppose that there is a "ready" node c between a "ready" node a and its direct neighbor, w.l.o.g. say the right neighbor b, which considers a also as its direct neighbor.

According to Section 3.3.2, a node goes from "ok" to "ready" only when it receives leases from its direct neighbors, which also believe it is their direct neighbor. Considering the position of node c's right neighbor[1] d, which must be different from node a or c, two cases are illustrated in Figure 6.3: either node d lies between a and b (*IncludeNeighbor*), or it lies farther to node c than b (*CrossNeighbor*). In both cases, node d grants the lease to c. Since the violation of the property *NeighborClosest* cannot go beyond these two situations, it is interesting to analyze them.

In order to detect violation as early as possible and on the other hand to make the invariant as strong as possible, the model checking analysis is conducted on nodes with all possible statuses except "dead", which is summarized as $NonDead \triangleq I \setminus DeadNodes$. Since all the nodes do not have specific status restriction, this information is omitted in Figure 6.3. After model checking it, if no violation is found, then this invariant is easier to prove because it only relies on changes of leaf set; if a violation is found, further analysis can be done to understand how this happens and if this may lead to a violation of property *NeighborClosest*.

By constructing the candidate invariant *IncludeNeighbor* as shown in Figure 6.3, notice that the right neighbor of node c cannot be a, nor b, because otherwise they will not consider each other as direct neighbors. Finally, the candidate invariant is specified in Property 6.2.3.

Property 6.2.3 (IncludeNeighbor)**.**

$$
\begin{array}{l}
IncludeNeighbor \triangleq \nexists a, b, c, d \in NonDead : \\
\quad \land\ c \neq d \land a \neq b \\
\quad \land\ RightNeighbor(lset[c]) = d \\
\quad \land\ LeftNeighbor(lset[d]) = c \\
\quad \land\ RightNeighbor(lset[a]) = b \\
\quad \land\ LeftNeighbor(lset[b]) = a \\
\quad \land\ CwDist(a, c) < CwDist(a, b) \\
\quad \land\ CwDist(a, d) < CwDist(a, b)
\end{array}
$$

After model checking *IncludeNeighbor* on HaeberlenPastry, the model checker TLC reports a counterexample as illustrated in Figure 6.4. Starting from the initial state when only node a is the "ready" node, nodes c and b concurrently join through a and a replies the "JoinRequest" to all of them as illustrated in Step 0 in Figure 6.4.

[1] The case for its left neighbor is symmetrical and hence omitted.

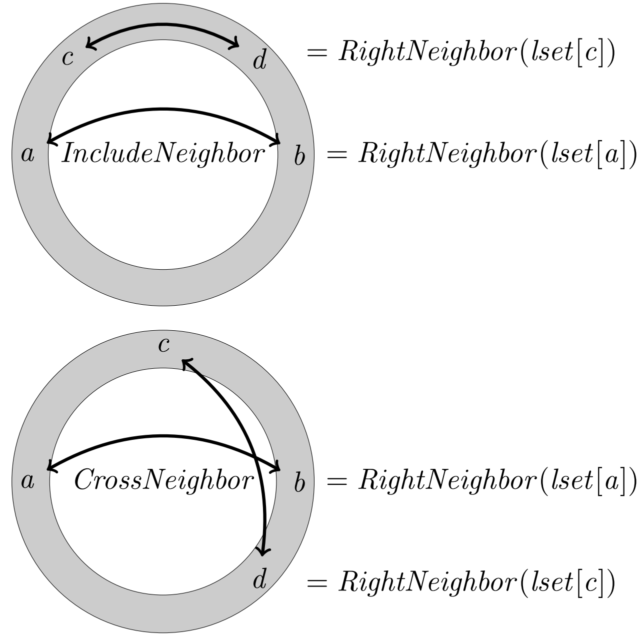

Figure 6.3: Intuition of hypothetical invariants *IncludeNeighbor* and *CrossNeighbor*.

Observe that node a does not add the joining nodes into its leaf sets and the "JoinReply" messages back to them contains only empty leaf sets of the node a. Staring at Step 0, it is intuitively hard to imagine that node a could consider node b as its right neighbor when both node c and d have already become "ready". The following trace breaks this intuition.

Node c first reacts on the "JoinReply" message from a and successfully becomes a "ready" node after probing process and lease granting process with node a as summarized in Step 1 in Figure 6.4.

Then node b receives the "JoinReply" messages from a, containing empty leaf sets. While node b adds a into its leaf sets and probes a back as illustrated in Step 2 in Figure 6.4, its probe message is delayed and not received by node a. Therefore, the leaf set of node a does not change in Step 2 comparing to previous step.

Meanwhile, node d also sends its "JoinRequest" to node a. It then receives the "JoinReply" message, containing the leaf sets of a, which includes node c. After it has probed c and a to complete its leaf sets, it has then got the leases from them and becomes "ready". By now, c, a, d are connected to each other and consider each other as neighbors according to their position respectively, as summarized in Step 3 in Figure 6.4.

Suddenly node a leaves the network and rejoins by sending a join message to c as shown in Step 4 in Figure 6.4.

It then receives the delayed "Probe" message from b and adds b into its leaf sets. By now, node a has ignored node c and d and reaches b as its right neighbor, which violates the invariant, as illustrated in the Step 5 in Figure 6.4.

One lesson learned from this counterexample is that the intuitive candidate invari-

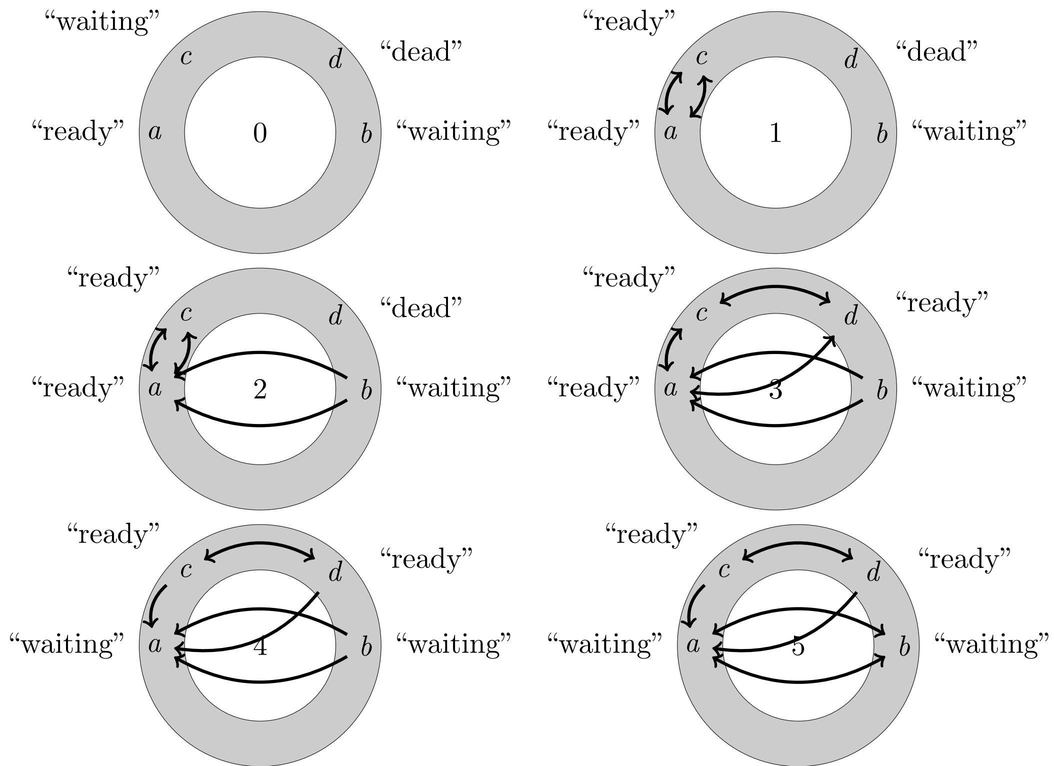

Figure 6.4: Counterexample of *IncludeNeighbor*.

ant *IncludeNeighbor* is too strong. Because this violation of *IncludeNeighbor* does not necessarily lead to a violation of property *NeighborClosest*, due to the fact that the node a still needs to wait for the "JoinReply" message from c.

Another lesson learned is that many intuition on a linear identifier space are broken in a network based on ring structure like Pastry. This violation shows how a node a can get its right neighbor far away from it ignoring the existing "ready" nodes c and d. In fact, the node a gets its right neighbor b from the other side of the ring, which is actually its left neighbor. This is due to the definition of $RightNeighbor(ls)$ in Definition 4.1.2.

The candidate invariant *CrossNeighbor* is also violated in a counter example replicating the rejoin problem illustrated in Figure 5.3 in Section 5.2.2. Therefore, the formal definition of *CrossNeighbor* and its analysis is omitted here.

All the previous counterexamples give the hint that the *churns*, namely departure and concurrent join of nodes, are the reasons of violation of the property *NeighborClosest*. After discovering many corner cases as counterexamples which are beyond of general intuition, it is not even sure at this point if the Pastry model without churn ensures *CorrectDelivery*. This motivates the following approach: making stronger assumptions of the Pastry model to first disable the problems discovered so far and see if the simplified model can be verified, then try to relax the assumption step by step.

6.3 A First Inductive Proof of NeighborClosest with Strong Assumptions

A first formal verification of Pastry model (IdealPastry) is completed with a rather strong assumption that not only presumes no departure of nodes, but also postulates no concurrent join between two "ready" nodes closed to each other as the example shown in Section 3.2.1.

This approach is described as follows.

6.3.1 Implementation of the Assumptions

The first assumption of IdealPastry is that no nodes leave the network, as defined in Assumption 3.4.3.

In CastroPastry and HaeberlenPastry, departure of a node i is formally specified as an action $NodeLeft(i)$, which changes the status of that node to "dead" and assigns all the other state variables to the initial empty values of a "dead" node. This action is not shown previously in this thesis in Chapter 3 because it is not a real action of a node, but only a simulation in formal model for silence departure of nodes. Besides, this action does not show up in LuPastry because that version assumes no departure of nodes.

Besides the action $NodeLeft(i)$, there are further actions to detect and recover the leaf sets of those nodes in CastroPastry and HaeberlenPastry, which are affected by the departure of node i. The handling of nodes departure together with those actions are illustrated in Figure 3.6 in Section 3.2.2.

In order to realize the assumption that no nodes leave the network, these actions simulating and handling nodes departure are simply removed from the disjunction *Next*. In fact, they do not show up in the Definition 4.2.1, where *Next* is defined for the final Pastry model, because this assumption is still kept till then.

The second assumption is defined in Assumption 6.3.1.

Assumption 6.3.1. *Not more than one node concurrently join between two "ready" nodes, which are the closest "ready" nodes to each other on the ring.*

In order to realize this assumption, a new state variable *toj* is introduced to remember to which node a "ready" node has replied a "JoinRequest" message. As long as the "ready" node replies a "JoinRequest" message to a particular node j in action RecJReq(i), it assigns $toj[i]$ with j. Then the action RecJReq(i) is disabled to react on any further "JoinRequest" message by adding precondition ensuring that $toj[i] = i$.

For preventing concurrent join through the other "ready" node, further preconditions are added that the *toj* value of the left and right neighbor of any "ready" node i must be set to themselves. Thus, no concurrent join could occur between the "ready" node and its neighbors. This violates the principle that a node in a distributed system can only read its local variables, but must use message passing to communicate with other nodes in the network. But on an abstract level of the model, node could request a "lock" from the neighbors, such that as soon as its own *toj* is occupied, its neighbors *toj* must be locked, until the local *toj* is freed and the node grants the lock. In fact, this is proposed by Ghodsi (2006). Of course, this is not a good solution, because for example, the concurrent joining on the right neighbor may request a lock of this node, which leads to a dead lock that can be only solved by random philosopher. Moreover, concurrent nodes joining on the right neighbor of the right neighbor of node i (which is different from node i and its left neighbor) may concurrently request the lock of *toj* on the right neighbor of node i. A consensus protocol needs to be designed to again resolve this concurrency problem. For this reason, this assumption is later relaxed in the thesis by improving the protocol and the improved version is verified to be correct and plausible to be implemented.

Since no concurrent join is allowed through the modification above, there is no need for the joining nodes to grant each other to become "ready". Hence, the precondition of the action RecLReq(i) is strengthened such that only "ready" nodes can grant leases.

After several modifications explained above on HAEBERLENPASTRY, model checker TLC is applied to validate the improved model using the techniques illustrated in Section 5.3. Then a stronger invariant *CompleteLeafSet* subsuming the invariant *NeighborClosest* is used as the inductive invariant and several corresponding invariants, such as *NoCCJoin*, *CoverToj* are defined with respect to the new assumptions, which will be illustrated in the coming section.

6.3.2 Invariants

The inductive invariants are extended further, in order to be strong enough to be proved themselves inductively for IDEALPASTRY, which is an earlier version of LUPASTRY. The

final conjunction of these invariants are illustrated in Definition 6.3.2

Definition 6.3.2 (*InvIdealPastry*).

$$
\begin{array}{l}
\wedge\ CompleteLeafSet \\
\wedge\ ReadyNeighborRight \\
\wedge\ ReadyNeighborLeft \\
\wedge\ HalfNeighborExt \\
\wedge\ NoCCJoin \\
\wedge\ NoCCJoinL \\
\wedge\ GrantNeighbor \\
\wedge\ GrantReady \\
\wedge\ LeaseGrant \\
\wedge\ CoverToJ \\
\wedge\ SingleReadyNodeToj \\
\wedge\ SingleReadyNode \\
\wedge\ LSnonEmpty \\
\wedge\ SemPRplyNonEmpty \\
\wedge\ SemJRplyLSnonEmpty \\
\wedge\ SemLReply \\
\wedge\ SemJRply \\
\wedge\ SemJoinNeighbor \\
\wedge\ SemJoinLeafSet \\
\wedge\ SemSingleReadyJoin \\
\wedge\ SemEmptyLSetSingleNode \\
\wedge\ SemJReq \\
\wedge\ JoinMsgDual \\
\wedge\ LSetInv \\
\wedge\ OneJRply \\
\wedge\ OneJReq \\
\wedge\ AtleastOneReady
\end{array}
$$

The *NeighborClosest* and *HalfNeighbor* are not components of *InvIdealPastry* anymore. The invariant *HalfNeighbor* is replaced with *HalfNeighborExt*, which is already explained and specified in Invariant 6.2.2 in Section 6.2.1. The invariant *NeighborClosest* is replaced by *CompleteLeafSet*, specified as follows in Invariant 6.3.3.

Invariant 6.3.3 (CompleteLeafSet).

$$
\begin{array}{l}
CompleteLeafSet \triangleq \forall i \in I, j \in ReadyNodes : \\
\quad lset[i] \neq EmptyLS(i) \wedge i \neq j \\
\quad \Rightarrow \wedge\ CwDist(i, RightNeighbor(i)) \leq CwDist(i, j) \\
\quad\quad \wedge\ CwDist(LeftNeighbor(i), i) \leq CwDist(j, i)
\end{array}
$$

In the definition of *CompleteLeafSet* the precondition of the implication states that the leaf sets of node i must be not empty. According to the Lemma 4.1.3, a node with

empty leaf sets has no neighbors. More precisely, it considers itself as its neighbor (on both sides). Therefore the precondition of *CompleteLeafSet* matches the cases of the other invariant *HalfNeighbor*, stating that either a node i has no neighbors, or there are other "ready" nodes. Besides, the conclusion of *CompleteLeafSet* is the same as property *NeighborClosest*. Since the property *HalfNeighbor* is already proved inductively before, the inductive proof of *NeighborClosest* is then reduced to the inductive proof of *CompleteLeafSet*. The following invariants are mainly used for supporting the inductive proof of *CompleteLeafSet*, which will be illustrated in Section 6.3.3.

The invariant *ReadyNeighborRight* specified in Invariant 6.3.4 states that if a node does not have an empty leaf sets (a not "dead" node or a node which has already received "JoinReply" message), then this node can not locate between a "ready" node and its right neighbor. Here it explicitly requires the node r to have no "ready" right neighbor, because otherwise this invariant subsumes the *CompleteLeafSet* and therefore can not be used as inductive hypothesis to prove the case when its right neighbor is not "ready".

Invariant 6.3.4 (ReadyNeighborRight)**.**

$$\begin{aligned} &\mathit{ReadyNeighborRight} \triangleq \forall r \in \mathit{ReadyNodes}, k \in I :\\ &\quad \land\ k \neq r\\ &\quad \land\ \mathit{lset}[k] \neq \mathit{EmptyLS}(k)\\ &\quad \land\ \mathit{RightNeighbor}(\mathit{lset}[r]) \notin \mathit{ReadyNodes}\\ &\quad \Rightarrow \mathit{CwDist}(r, \mathit{RightNeighbor}(\mathit{lset}[r])) \leq \mathit{CwDist}(r, k) \end{aligned}$$

The invariant *ReadyNeighborLeft* is the symmetrical invariant to *ReadyNeighborRight* referring to the left neighbor instead of the right neighbor. These two invariants are intuitively confirming the precondition of the action RecJReq(i), which implements the assumption that no nodes are joining concurrently among the region of a "ready" node and its left and right neighbors.

The invariant *NoCCJoin* specified in Invariant 6.3.5 states that if there is a node k between a "ready" node i and its right neighbor r, which is "ready" too, then node k is either joined by node i (in this case, there is no nodes trying to join by node r), or node k is joined by node r (in that case, no node is trying to join by i).

Invariant 6.3.5 (NoCCJoin)**.**

$$\begin{aligned} &\mathit{NoCCJoin} \triangleq \forall i, r \in \mathit{ReadyNodes}, k \in I :\\ &\quad \land\ i \neq k\\ &\quad \land\ \mathit{lset}[k] \neq \mathit{EmptyLS}(k)\\ &\quad \land\ r = \mathit{RightNeighbor}(\mathit{lset}[i])\\ &\quad \land\ i \neq r\\ &\quad \land\ \mathit{CwDist}(i, k) < \mathit{CwDist}(i, r)\\ &\quad \Rightarrow \lor\ \mathit{toj}[i] = k \land \mathit{toj}[r] = r\\ &\qquad\ \ \lor\ \mathit{toj}[r] = k \land \mathit{toj}[i] = i \end{aligned}$$

The invariant *NoCCJoinL* is the symmetrical invariant to *NoCCJoin*, but replace the right neighbor with left neighbor and changes the corresponding positions. *NoCCJoin*

and *NoCCJoinL* together formulate directly the assumption as invariants that no concurrent nodes are joining among the region of a "ready" node and its left and right neighbors.

The invariant *CoverToJ* specified in Invariant 6.3.6 states that if a "ready" node r covers a node k whose leaf sets are not empty, then node k must be joining by r.

Invariant 6.3.6 (CoverToJ)**.**

$$\begin{aligned} &\mathit{CoverToJ} \triangleq \forall r \in \mathit{ReadyNodes}, k \in I : \\ &\quad \land\ r \neq k \\ &\quad \land\ \mathit{lset}[k] \neq \mathit{EmptyLS}(k) \\ &\quad \land\ \mathit{Covers}(\mathit{lset}[r], k) \\ &\ \Rightarrow \mathit{toj}[r] = k \end{aligned}$$

The invariant *CoverToj* clarifies the intuition of the link between the coverage of a "ready" node and its value of the state variable *toj*, namely, as long as the "ready" node covers some other node having not empty leaf sets and currently joining through the "ready" node. This also confirms the precondition of the action RecJReq(i) that a "ready" node only replies a "JoinRequest" message from a node which it covers.

The invariants *GrantNeighbor*, *GrantReady* and *LeaseGrant* state the properties of state variables *lease* and *grant*. The invariant *GrantNeighbor* specified in Invariant 6.3.7 ensures that if node i is still not yet "ready", which already gets lease granted by a "ready" node k, then node i must be either the left or the right neighbor of node k.

Invariant 6.3.7 (GrantNeighbor)**.**

$$\begin{aligned} &\mathit{GrantNeighbor} \triangleq \forall k \in \mathit{ReadyNodes}, i \in I : \\ &\quad \land\ i \notin \mathit{ReadyNodes} \\ &\quad \land\ i \in \mathit{grant}[k] \\ &\ \Rightarrow \lor\ i = \mathit{LeftNeighbor}(\mathit{lset}[k]) \\ &\qquad \lor\ i = \mathit{RightNeighbor}(\mathit{lset}[k]) \end{aligned}$$

The invariant *GrantNeighbor* confirms the transition defined in action RecLReq(i), that node k only grants the lease to its left or right neighbor. From another perspective, this invariant also states that the only possibility for a node k in $\mathit{grant}[i]$ not to be the neighbor of node i, is that this node k is already a "ready" node, despites the fact that it has got the lease granted from i. Hence, the invariant *GrantNeighbor* also helps to give intuition of the state variable *grant*, namely, this variable marks the history to which nodes a node has ever granted the leases.

The invariant *GrantReady* (Invariant 6.3.8) states that the nodes which has granted other nodes must be a "ready" node.

Invariant 6.3.8 (GrantReady)**.**

$$\mathit{GrantReady} \triangleq \forall i, j \in I : j \in \mathit{grant}[i] \Rightarrow i \in \mathit{ReadyNodes}$$

This invariant confirms the precondition of the action RecLReq(i) that only “ready” nodes can grant.

The invariant *LeaseGrant* (Invariant 6.3.9) states that if a node n is in another node i’s lease, then i must be already granted by n, which gives the intuition of the relationship between state variable *lease* and *grant*.

Invariant 6.3.9 (LeaseGrant)**.**

$$LeaseGrant \triangleq \forall i, n \in I : n \in lease[i] \Rightarrow i \in grant[n]$$

Through the invariants *GrantReady* and *LeaseGrant*, a node i can infer from its received leases that the nodes in its *lease* are “ready” nodes.

The invariants with names consisting of *Sem* and the message names (e.g. *SemLReply*) state in fact the preconditions ($status[n]$ = “ready” is equivalent to $n \in$ ReadyNodes) of the actions (RecLReq(i) specified in Definition 4.2.14), which generate those messages (“LeaseRequest” message), and their postconditions ($grant'[n] = grant[n] \cup \{m.mreq.node\}$) occurring together with the generation of the messages. For example, the invariant *SemLReply* as specified as follows in Invariant 6.3.10 states that when a node i receives “LeaseReply” message from n, then node n must be a “ready” node and if the “LeaseReply” message grants the lease, then i is already added into the $grant[n]$. Other message types also have similar invariants like *SemLReply*.

Invariant 6.3.10 (SemLReply)**.**

$$\begin{aligned}
SemLReply \triangleq\ & \forall i, n \in I, m \in receivedMsgs : \\
& \quad \wedge\ m.mreq.type = \text{“}LeaseReply\text{”} \\
& \quad \wedge\ m.mreq.lset.node = n \\
& \quad \wedge\ m.destination = i \\
& \Rightarrow \wedge\ n \in ReadyNodes \\
& \quad\ \ \wedge\ (m.mreq.grant = \text{TRUE} \Rightarrow i \in grant[n])
\end{aligned}$$

Further invariants state the corner cases. For example, *SingleReadyNodeToj* states that if there is only one Ready node r on the ring (and it is empty), then any node k whose leaf sets are not empty must be joining by r.

6.3.3 Proof Sketch of CompleteLeafSet as an Example

With the general understanding of the invariants, the inductive proof of Invariant *CompleteLeafSet* is explained in the following sketch.

Similar to the methodology used for the inductive proof of Invariant *HalfNeighbor* illustrated in Section 6.2.1, violation of the proving target *CompleteLeafSet* lies in changes of the two variables *lset* and *status*.

With respect to the change of *lset*, only actions extending leaf sets need to be considered, because no actions remove a node from any leaf sets according to the assumptions and modification of Pastry model explained in Section 6.3.1.

For further proof, an additional lemma on the operation $AddToLSet(ls, delta)$ (specified in Definition 4.1.2 in Section 4.1.2) is introduced here to demonstrate how lemmas on data structures are added aside the inductive proof to be used as argument of several proof steps. This lemma states that after adding new nodes into leaf sets, the new direct neighbors are always closer to the node than before, or they remain the same.

Lemma 6.3.11 (AddToLSetInvCo)**.**

$$
\begin{array}{l}
AddToLSetInvCo \triangleq \forall\ s, ls \in LSet, d \in \text{SUBSET} I :\\
\quad ls = AddToLSet(d, s)\\
\quad \Rightarrow \land\ CwDist(LeftNeighbor(ls), ls.node)\\
\qquad\qquad \leq CwDist(LeftNeighbor(s), s.node)\\
\qquad \land\ CwDist(ls.node, RightNeighbor(ls))\\
\qquad\qquad \leq CwDist(s.node, RightNeighbor(s))
\end{array}
$$

The lemma is easy to prove based on the definition of $AddToLSet(ls, delta)$ and the definition of $LeftNeighbor(ls)$ (also specified in Definition 4.1.2 in Section 4.1.2). Intuitively, if the added node is farther than the previous direct neighbor, the direct neighbor does not change after the node is added, because $AddToLSet(ls, delta)$ always choose the minimal nodes to be the new leaf sets.

By induction hypothesis of invariant *CompleteLeafSet* (see Invariant 6.3.3) and the lemma *AddToLSetInvCo*, together with transitivity of $\leq$ on distances, it can be deduced that the invariant *CompleteLeafSet* holds when leaf sets are changed.

The proof gets more complicated with respect to the changes of state variable *status*, when a joining node turns from “ok” to “ready”. This happens when the node receives a “LeaseReply” message and the flag *okToReady* is TRUE in the action RecLReply(i) as specified in Definition 4.2.15. The idea is to construct the possible violation of the Invariant *CompleteLeafSet* and then to prove that this violation can not occur by contradiction.

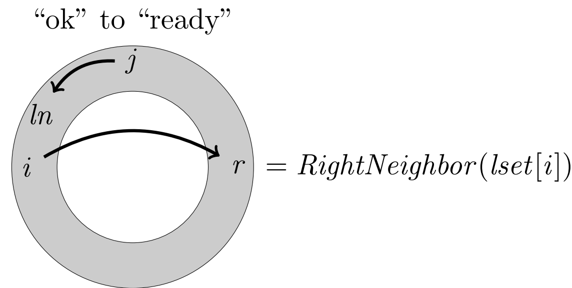

Figure 6.5: Hypothetical violation of *CompleteLeafSet* by its inductive proof.

The violation case is illustrated in Figure 6.5. As the first assumption of the proof, node j, which is “ok” and going to become “ready” by the action RecLReply(j), locates between an arbitrary node i, whose leaf sets are not empty, and its direct right neighbor r. The next task is to find out the reason why this configuration can not occur. The

idea is to analyze the position of the direct left neighbor of node j, named as ln. If it can be proved that ln lies between node i and node r, in the upper half of the ring in Figure 6.5, then the induction hypothesis of *CompleteLeafSet* can be used to contradict this situation. Hence, it must be shown that node ln is "ready" and it lies between node i and its direct neighbor r, which breaks the proof to two sub-proofs.

The first part can be proved using the invariants *SemLReply* (see Invariant 6.3.10), *LeaseGrant* and *GrantReady* (both introduced in Section 6.3.2) contained in hypothesis *InvIdealPastry* and the precondition of action RecLReply(j). It is easy to deduce that the node j is granted by its left neighbor, based on the fact that knowing from the local variable *okToReady* in RecLReply(j), the left neighbor has either already granted the lease before because it is in the *lease*[j] (using *LeaseGrant*), or it has just granted the lease through "LeaseReply" message (using *SemLReply*). Then using *GrantReady*, the left neighbor of the node i can be derived to be "ready", because it has granted node i as shown before.

The second sub-proof needs to use *ReadyNeighborRight* specified in Invariant 6.3.4, which deduces that the distance from a "ready" node ln to its right neighbor j is smaller than from ln to some arbitrary node i, whose leaf sets are not empty. This statement is equivalent to saying that ln lies between i and j as shown in Figure 6.5. Besides, it is given as proof assumption at the beginning of the proof that j lies between i and r, then the final conclusion can be derived by transitivity of $\leq$ of distances of nodes on the ring. However, in order to use *ReadyNeighborRight*, the preconditions according to its definition must be fulfilled. Node ln is proved to be "ready" by the first part illustrated above. Moreover, it is to show that node j is the right neighbor of node ln.

The idea is to use the Invariant *GrantNeighbor* (as defined in Invariant 6.3.7). It shows that node j, which is granted by node ln, is indeed the right neighbor of node ln, because node j is not yet "ready" according to the precondition of the action RecLReply(j).

Further corner cases are omitted here such that this proof sketch can focus on the basic ideas and intuition. A detailed proof encoded in TLA$^+$, which is automatically verified through theorem prover TLAPS, is available online at VeriDis (2013).

This proof sketch illustrates how the invariants are used to deduce that the invariant *CompleteLeafSet* is valid focusing on the critical transitions where the proof goal might be violated. The invariants, which are not used in this proof, are necessary for proving those invariants used here, such that all together as a conjunction *InvIdealPastry* can be used as inductive invariant to prove throughout all possible actions of Pastry.

6.4 Relaxing the Assumptions

The verified Pastry model shown in the previous section has limitations due to the two strong assumptions that nodes do not leave the network and concurrent join of nodes between two neighboring "ready" nodes does not occur. The following task is to relax all possible assumptions so far to see if it is still able to be verified.

6.4.1 The Problem of Allowing Departure of Nodes

In order to allow departure of nodes, the actions *NodeLeft*(i) and corresponding actions from CastroPastry and HaeberlenPastry, such as SuspectFaulty(i, j), TimeOut(i, j), are added into the possible actions of Pastry model *Next* of the earlier version of LuPastry, so that nodes can freely leave the network. Then a manual constructed counterexample shows a possible separation of the network, which motivates the reservation of this assumption.

Separation of the Network due to Concurrent Departures of Nodes

If the many nodes can drop the network concurrently, a separation of the network could occur, as mentioned in Haeberlen et al. (2005). However, no effective solution is found to ensure the property *CorrectDelivery*. The violation of the property *CorrectDelivery* is illustrated in Figure 6.6.

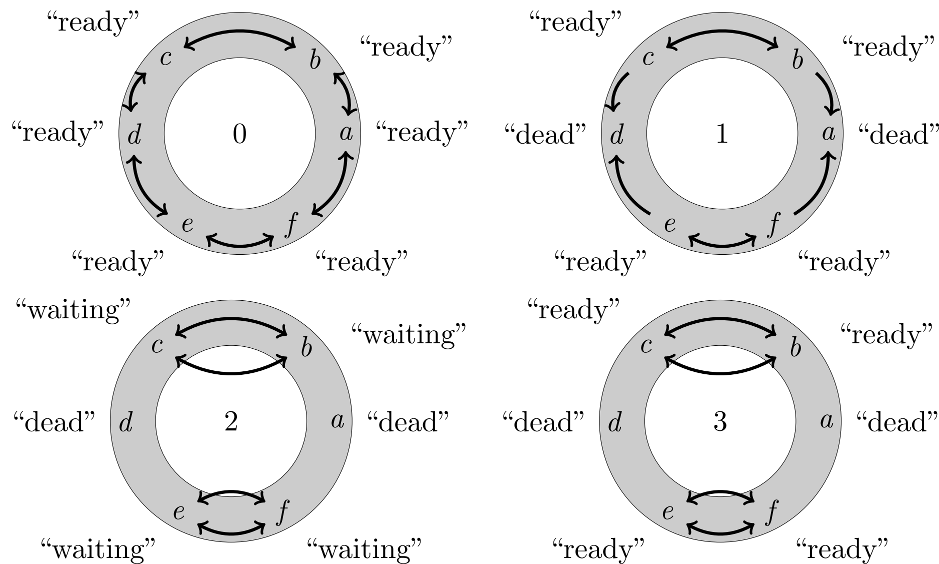

Figure 6.6: Separation of the network due to concurrent departures of nodes.

Initially, there are six "ready" nodes in the network as shown in the Step 0 in Figure 6.6. Remark that the length of the leaf sets L is one.

Then nodes d and a concurrently drop off, whereas their direct neighbors still have them in their leaf sets as illustrated in Step 1 in Figure 6.6.

After a while, when the leases of them are expired by their neighbors, they use the action *SuspectFaulty*(i, j) to detect the "dead" nodes and when waiting time is expired, they remove them from the leaf sets, as illustrated in Step 2 in Figure 6.6.

Subsequently, the rest of the nodes start to repair the leaf sets by probing the current members in their leaf sets, which simply answer the "Probe" messages to each other without any additional information. As a result, they turn to "ok" in the action

RecPRply(i) when receiving the "ProbeReply" messages from each other. In the next transitions, nodes request the leases from each other and get grants, which make them "ready", as illustrated in the Step 3 in Figure 6.6.

Now nodes c and b builds a separate network from the one consisting e and f. Such kinds of separation problems of Pastry network motivate that the Assumption 3.4.3 mentioned in Section 3.4 should be kept that no nodes leave the network. In fact, this result is already mention in several literatures, such as Castro et al. (2004), Haeberlen et al. (2005), Ghodsi (2006) and Zave (2012), but no solution is proposed to prevent network separation with guarantee.

6.4.2 The Approach of Allowing Concurrent Join

The precondition of the action RecJReq(i) is not realistic as discussed for Assumption 6.3.1. The first approach is to remove this unrealistic precondition. This has a consequence that concurrent join of nodes between neighboring "ready" nodes may occur.

Another modification of the Pastry model is relaxing the precondition of action RecLReq(i) from "ready" nodes to "ready" or "ok" nodes. Before the modification, the precondition requires that only "ready" nodes can handle the "LeaseRequest" messages. Now since concurrent nodes are joining between neighboring "ready" nodes, they need each other to grant the leases, in order to complete the lease granting process and become "ready", otherwise they would have to wait in a live lock in status "ok", which prevents any of them to join successfully.

The modifications lead to another intermediate version between HaeberlenPastry and LuPastry, which is then improved based on the violation discovered as follows.

Pure Concurrent Join Problem

The modifications of the protocol leads to a violation of the property *NeighborClosest*, which is already illustrated in Section 3.2.4 and here it is explained in more details. Remark that this corner case is impossible to detect using the model checker without specific selection of node identifiers as illustrated in the Figure 6.7.

Assume there are five nodes a, b, c, d and e on the network as shown in Step 0 in Figure 6.7. Only the two nodes a and b are initially "ready". The rest of the nodes are initially "dead". The leaf set length L is one, meaning each node has only one node in each side of its leaf sets.

First, node e joins the network through node a and receives the "JoinReply" message from node a, which contains b in its leaf sets. But the probing message from node e to node a is delayed. Only node b has received the "Probe" message from e. Node b now has a new left neighbor, node e. The result of these transitions is illustrated in Step 1 in Figure 6.7.

Then node d joins the network through b. It gets the "JoinReply" message from node b containing node a and e. It finds that the candidate leaf set members are e and b, which are its left and right neighbors. Since the length of the leaf sets is one, node a

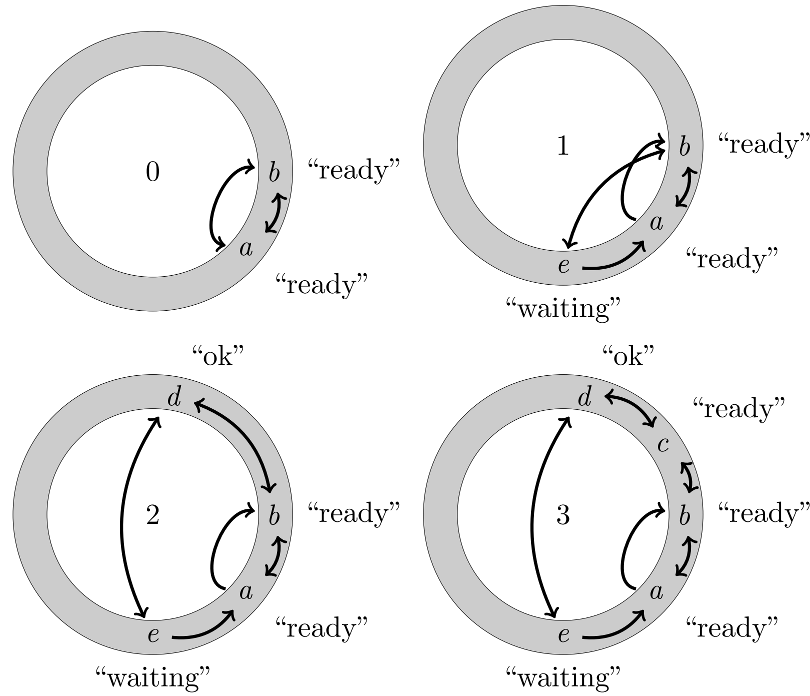

Figure 6.7: Concurrent join with 5 nodes.

is excluded in the candidate leaf set members. Therefore, node d only probes both node e and node b and becomes "ok" after both of nodes e and b reply the "Probe" messages. Observe that node e can reply such "Probe" messages because it is a "waiting" node and has leaf set members b and a before it adds e into its leaf sets. The result of these transitions is illustrated in Step 2 in Figure 6.7.

Now node c joins the network through b. It gets the "JoinReply" message from node b, which contains the node d. Then node c probes node d and b and becomes "ok". Subsequently, it requests the leases from the neighbors. Since d and b are "ok" or "ready", they both grant the leases to c, which makes c "ready". The result of these transitions is illustrated in Step 3 in Figure 6.7. Now the "ready" node a, which considers its right and left neighbor to be the identical node b, has ignored the existence of "ready" node c between itself and its right neighbor b, which violates the invariant *NeighborClosest*.

In this counterexample, no node leaves the network and only join of nodes is allowed. The invariant *NeighborClosest* is violated as shown in Figure 6.7 if there are many nodes concurrently joining the network through one node b. One of the solutions is to restrict the number of nodes one "ready" node can handle, such that if new nodes like d and c come to join through b when it has already started to help e to join, they have to wait until the joining node e has completed its join process through the "ready" node b.

For simplicity, the new assumption as mentioned in Assumption 3.4.4 in Section 3.4 allows a "ready" node to handle at most one joining node. This modification still allow

concurrent join between two neighboring "ready" nodes. After this restriction, node c cannot join the network and the concurrent joining node e and d can turn to "ready" only after node a has replied the "Probe" message to e, which enables e to become "ok".

6.4.3 The Meaning of Allowing Message Loss

Besides the two limitations discussed before, it is considered to relax further assumptions. According to Assumption 3.4.2, message corruption and message loss are both excluded from Pastry. Message corruption relates to security properties as mentioned before, therefore no relaxing is possible. However, message loss can be considered as possible action of Pastry model. In fact, message loss is modeled as an action in TLA$^+$, which simply removes arbitrary messages from the message pool *receivedMsgs*. The effect of such action is that some joining node might not be able to continue the join process due to the loss of crucial message such as "JoinReply" message.

Adding the action *MessageLoss*() into the *Next* formula in TLA$^+$ does not affect the final proof of safety property *NeighborClosest* and *CorrectDelivery*. But this does not really bring too much novelty because on the one hand, there is no mechanism to rescue the lost messages in Pastry, on the other hand, ensuring that a node can eventually join into the network despite of message loss is in fact a liveness property, which is beyond the verification purpose of this thesis.

6.4.4 Modification of the Proof and Invariants

After modifications of the actions of Pastry model from the version verified for *CompleteLeafSet* to LuPastry, one could think of modifying the invariants based on the understanding of assumptions and the invariants. But since there are too many invariants, there is a more efficient methodology than analyzing them one after another. This methodology can get the most benefits from the previous proof.

In order to prevent the restart of discovery of invariant, the previous formal proof is first rechecked automatically using TLAPS, which then automatically discovers which parts of the proof are no longer valid due to those modifications.

When TLAPS reports the invalid proof obligations, the following questions are proposed subsequently:

- Why is this part of proof no more valid?
- Should this invariant be modified and how?
- Is this modified invariant valid?
- Is this modified invariant still useful to prove other invariants as before?

For each discovered corresponding invariant as proof goal, analysis is conducted based on the questions one after another.

Firstly, the broken down proof is analyzed by checking if the preconditions of the proof is no longer valid due to modification of the protocol at a particular action or if

the invariant as proof obligation is no longer valid. For the first case, a new proof is to constructed adapting the modification of the protocol case by case. For the latter case, the invariant is analyzed by intuition and the model checker TLC is applied to discover a counterexample to confirm the intuition and help reformulate the invariant. For example, the previous invariant *GrantReady* as specified in Invariant 6.3.8 is no longer valid by intuition because it reflects exactly the modification of the protocol on the action RecLReq(i) that not only "ready" nodes can handle "LeaseRequest" messages, but also "ok" nodes.

Secondly, two perspectives according to the last two questions are considered to modify the previous invariants that are no longer valid or to construct the new candidates for invariants to support the proof of those valid ones. On one hand, the proposed candidate invariant should be valid, in the sense that it first should pass the model checking for the case of four instances in the network, and then be formally proved at the same place where the proof for the previous invariant is broken. On the other hand, applying this proposed candidate should be still able to infer the previous proof obligations as sub-proof for other invariants.

For example, the previous invariant *GrantReady* can be reformulated according to its violation, such that a new invariant *GrantOK* can be constructed as followed:

Invariant 6.4.1 (GrantOK)**.**

$$GrantOK \triangleq \forall i, j \in I : j \in grant[i] \Rightarrow i \in ReadyOK \wedge j \in ReadyOK$$

This new invariant should be valid according to the intuition and it is also easy to prove. But it is no more useful for proving *CompleteLeafSet* according to the proof in Section 6.3.3, where *GrantReady* is used to infer that the left neighbor granting the joining node is "ready", such that inductive hypothesis of *CompleteLeafSet* can be applied on the "ready" left neighbor. Hence, the proof of *CompleteLeafSet* needs to be modified: either another proof approach should be found, in which other invariants for "ready" nodes can be applied instead of the previous *GrantReady*, or the invariant *CompleteLeafSet* can be strengthen, such that it applies on not only "ready" nodes but also "ok" nodes, in order to make use of the invariant *GrantOK*. Since the model checker produces counterexamples of strengthened *CompleteLeafSet* on "ok" and "ready" nodes, the proof approach for *CompleteLeafSet* must be changed. However, the invariant *GrantOK* is still useful for proving other invariants.

Using the methodology described above, half the invariants need modification due to their violations or inadequacies for supporting desired proofs as arguments. The results are summarized in the next section.

6.5 Final Invariants and the Proof

6.5.1 Invariants

Based on the Assumption 3.4.3 stating that no nodes leave the network and the Assumption 3.4.4 stating that a "ready" node can at most handle one joining node at a

time, the property *NeighborClosest* (Property 4.3.7) can be further reduced to rather strong invariants *IRN* and *NRI* in the Pastry model LuPastry. These two invariants together subsumes the property *NeighborClosest*. The difference is that *NeighborClosest* guarantees that "ready" nodes do not ignore other "ready" nodes between themselves and their neighbors, while *IRN* and *NRI* states that every node does not ignore any "ready" node between itself and its neighbor.

Invariant 6.5.1. *There cannot be a Ready node closer to i, than its left and right neighbors.*

$$
\begin{aligned}
&IRN \triangleq \forall i \in I, r \in ReadyNodes : i \neq r \\
&\quad \Rightarrow CwDist(i, RightNeighbor(lset[i])) \leq CwDist(i, r) \\
&NRI \triangleq \forall i \in I, r \in ReadyNodes : i \neq r \\
&\quad \Rightarrow CwDist(LeftNeighbor(lset[i]), i) \leq CwDist(r, i)
\end{aligned}
$$

The proof of *IRN* will be sketched at the end of this section, which needs other invariants shown in the *InvLuPastry* as follows.

Definition 6.5.2 gives the complete list of inductive invariants used for proving *NeighborClosest* for the final model of Pastry. Comparing to the invariants *InvIdealPastry* defined before in Definition 6.3.2, some invariants, such as *OnePRply* and *OneJReq*, remain the same and are still valid for the new Pastry model. Many invariants are no longer valid, such as *GrantReady* illustrated in Section 6.4.4, which is then changed to *GrantOK* defined in Invariant 6.4.1.

As follows, each of the invariants will be illustrated with an intuitive introduction and its formal definition.

Invariant 6.5.3. *If a node i has already got the lease granted (see Section 3.2.2) by another node k, then neither of these two nodes have empty leaf sets. The other node cannot be closer to them than their neighbors.*

$$
\begin{aligned}
&GrantNeighborNew \triangleq \forall k, i \in I : \\
&\quad \land\ i \neq k \\
&\quad \land\ i \in grant[k] \\
&\quad \Rightarrow \land\ lset[i] \neq EmtyLS(i) \\
&\quad\quad \land\ lset[k] \neq EmptyLS(k) \\
&\quad\quad \land\ CwDist(LeftNeighbor(lset[i]), i) \leq CwDist(k, i) \\
&\quad\quad \land\ CwDist(i, RightNeighbor(lset[i])) \leq CwDist(i, k) \\
&\quad\quad \land\ CwDist(LeftNeighbor(lset[k]), k) \leq CwDist(i, k) \\
&\quad\quad \land\ CwDist(k, RightNeighbor(lset[k])) \leq CwDist(k, i)
\end{aligned}
$$

Invariant 6.5.4. *If a not yet "ready" node i lies between two other different nodes l and r, and node i is joined through one of the node (e.g. l), whereas this node (i.e. l) has granted its lease to the other node (i.e. r), then the direct neighbor of i must be closer*

to i than the other node (i.e. r).

$$
\begin{aligned}
&\mathit{GrantHistR} \triangleq \forall l, i, r \in I : \\
&\quad \land\ \mathit{toj}[l] = i \\
&\quad \land\ i \neq r \\
&\quad \land\ r \in \mathit{grant}[l] \\
&\quad \land\ i \notin \mathit{ReadyNodes} \\
&\quad \land\ \mathit{CwDist}(i, r) < \mathit{CwDist}(l, r) \\
&\Rightarrow \mathit{CwDist}(i, \mathit{RightNeighbor}(\mathit{lset}[i])) \leq \mathit{CwDist}(i, r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{GrantHistL} \triangleq \forall l, i, r \in I : \\
&\quad \land\ \mathit{toj}[r] = i \\
&\quad \land\ i \neq l \\
&\quad \land\ l \in \mathit{grant}[l] \\
&\quad \land\ i \notin \mathit{ReadyNodes} \\
&\quad \land\ \mathit{CwDist}(l, i) < \mathit{CwDist}(l, r) \\
&\Rightarrow \mathit{CwDist}(\mathit{LeftNeighbor}(\mathit{lset}[i]), i) \leq \mathit{CwDist}(l, i)
\end{aligned}
$$

Invariant 6.5.5. *Only Ready or OK node can grant the leases.*

$$\mathit{GrantOK} \triangleq \forall i, j \in I : j \in \mathit{grant}[i] \land j \neq i \Rightarrow i \in \mathit{ReadyOK} \land j \in \mathit{ReadyOK}$$

Invariant 6.5.6. *A "ready" node never covers another node, which has joined through some other node.*

$$
\begin{aligned}
&\mathit{CoverNoNode} \triangleq \forall i \in \mathit{ReadyNodes}, j, k \in I : \\
&\quad \land\ \mathit{Covers}(\mathit{lset}[i], k) \\
&\quad \land\ i \neq k \\
&\quad \land\ j \neq k \\
&\Rightarrow \mathit{toj}[j] \neq k
\end{aligned}
$$

Invariant 6.5.7. *If a node i has joined through some node l, then no other node which holds a grant from l can cover node i.*

$$
\begin{aligned}
&\mathit{CoverNoToj} \triangleq \forall l, i, r \in I : \\
&\quad \land\ \mathit{Covers}(\mathit{lset}[r], i) \\
&\quad \land\ r \in \mathit{grant}[l] \\
&\quad \land\ \mathit{toj}[l] = i \\
&\quad \land\ i \neq r \\
&\quad \land\ r \neq l \\
&\Rightarrow \text{FALSE}
\end{aligned}
$$

Invariant 6.5.8. *If a not yet "ready" node i has joined through some node $r1$, then between these two nodes, there exists no node k different from i, which has joined through some node $r2$ different from $r1$. Note that the node $r2$ different from $r1$ is not explicit*

in the following predicate; it is implied by the parts that assert $i \neq k$*,* $toj[r1] = i$ *and* $toj[r2] = k$*.*

$$
\begin{aligned}
&TojClosestL \triangleq \forall r1, r2, i, k \in I : \\
&\quad \land\ i \neq r1 \\
&\quad \land\ i \neq k \\
&\quad \land\ toj[r1] = i \\
&\quad \land\ RightNeighbor(lset[r1]) = i \\
&\quad \land\ i \notin ReadyNodes \\
&\quad \land\ toj[r2] = k \\
&\quad \land\ r2 \neq k \\
&\Rightarrow CwDist(r1, i) \leq CwDist(k, i)
\end{aligned}
$$

$$
\begin{aligned}
&TojClosestR \triangleq \forall r1, r2, i, k \in I : \\
&\quad \land\ i \neq r1 \\
&\quad \land\ i \neq k \\
&\quad \land\ toj[r1] = i \\
&\quad \land\ LeftNeighbor(lset[r1]) = i \\
&\quad \land\ i \notin ReadyNodes \\
&\quad \land\ toj[r2] = k \\
&\quad \land\ r2 \neq k \\
&\Rightarrow CwDist(i, r1) \leq CwDist(i, k)
\end{aligned}
$$

Invariant 6.5.9. *If the leaf sets of some not yet "ready" node* i *is not empty, then there must exist a "ready" node, through which node* i *has joined the network.*

$$
\begin{aligned}
&TojNoReady \triangleq \forall i \in I : \\
&\quad \land\ i \notin ReadyNodes \\
&\quad \land\ lset[i] \neq EmtyLS(i) \\
&\Rightarrow \exists r \in ReadyNodes : toj[r] = i
\end{aligned}
$$

Invariant 6.5.10. *If a not yet "ready" node* i *has joined through some node* r*, then node* i *must be* r*'s direct neighbor.*

$$
\begin{aligned}
&SemToj \triangleq \forall r, i \in I : \\
&\quad \land\ i \notin ReadyNodes \\
&\quad \land\ toj[r] = i \\
&\quad \land\ r \neq i \\
&\Rightarrow \lor\ RightNeighbor(lset[r]) = i \\
&\quad\ \ \lor\ LeftNeighbor(lset[r]) = i
\end{aligned}
$$

Invariant 6.5.11. *If a not yet "ready" node* i *has joined through some node* r*, and node* i *is* r*'s direct neighbor on one side, then the node* r *must be the direct neighbor of* i *to*

the other side. [2]

$$
\begin{aligned}
TojDualR \triangleq\ & \forall r, i \in I : \\
& \land\ r \neq i \\
& \land\ toj[r] = i \\
& \land\ RightNeighbor(lset[r]) = i \\
& \land\ i \notin ReadyNodes \\
& \land\ lset[i] \neq EmtyLS(i) \\
\Rightarrow\ & LeftNeighbor(lset[i]) = r
\end{aligned}
$$

$$
\begin{aligned}
TojDualL \triangleq\ & \forall r, i \in I : \\
& \land\ r \neq i \\
& \land\ toj[r] = i \\
& \land\ LeftNeighbor(lset[r]) = i \\
& \land\ i \notin ReadyNodes \\
& \land\ lset[i] \neq EmtyLS(i) \\
\Rightarrow\ & RightNeighbor(lset[i]) = r
\end{aligned}
$$

Invariant 6.5.12. *Only "ready" node can allow a new node to join.*

$$TojReady \triangleq \forall r \in I : toj[r] \neq r \Rightarrow r \in ReadyNodes$$

Invariant 6.5.13. *Different nodes must be joined through different nodes.*

$$
\begin{aligned}
TojInjective \triangleq\ & \forall i, j, k \in I : \\
& \land\ toj[i] = k \\
& \land\ toj[j] = k \\
& \land\ i \neq k \\
& \land\ j \neq k \\
\Rightarrow\ & i = j
\end{aligned}
$$

Invariant 6.5.14. *If a node i lies in another node's leaf sets, then either it has just joined through that node, or it already has some node in its own leaf sets.*

$$
\begin{aligned}
LSnonEmptyToj \triangleq\ & \forall i, j \in I : \\
& \land\ i \in GetLSetContent(lset[j]) \\
& \land\ i \neq j \\
\Rightarrow\ & \lor\ lset[i] \neq EmptyLS(i) \\
& \lor\ toj[j] = i
\end{aligned}
$$

Invariant 6.5.15. *As long as there is a "JoinReply" message to node j, there can never be a "ready" node, which lies between the future left and right neighbor of node j, which will be then initiated after the node j has received the "JoinReply" message. If the node j considers i as its left neighbor, then its right neighbor will be the current right neighbor*

[2] This is in fact an extension to the invariant 6.5.10 (*SemToj*), in order to avoid too complex formula structure of the invariants for better readability and simpler proof structure.

of the node i. If the node j considers i as its right neighbor, then it will have the same left neighbor as the node i.

$$
\begin{aligned}
&\mathit{JoinNCL} \triangleq \forall m \in \mathit{receivedMsgs}, r \in \mathit{ReadyNodes}, i, j \in I : \\
&\quad \land\ m.mreq.type = \text{“JoinReply”} \\
&\quad \land\ m.mreq.lset.node = i \\
&\quad \land\ m.destination = j \\
&\quad \land\ r \neq i \\
&\quad \land\ \mathit{LeftNeighbor}(\mathit{AddToLSet}(\mathit{GetLSetContent}(m.mreq.lset), \mathit{EmptyLS}(j))) = i \\
&\quad \Rightarrow \mathit{CwDist}(i, \mathit{RightNeighbor}(m.mreq.lset)) \leq \mathit{CwDist}(i, r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{JoinNCR} \triangleq \forall m \in \mathit{receivedMsgs}, r \in \mathit{ReadyNodes}, i, j \in I : \\
&\quad \land\ m.mreq.type = \text{“JoinReply”} \\
&\quad \land\ m.mreq.lset.node = i \\
&\quad \land\ m.destination = j \\
&\quad \land\ r \neq i \\
&\quad \land\ \mathit{RightNeighbor}(\mathit{AddToLSet}(\mathit{GetLSetContent}(m.mreq.lset), \mathit{EmptyLS}(j))) = i \\
&\quad \Rightarrow \mathit{CwDist}(\mathit{LeftNeighbor}(m.mreq.lset), i) \leq \mathit{CwDist}(r, i)
\end{aligned}
$$

Invariant 6.5.16. *As long as there is a “JoinReply” message to node j, the sender of this message will never grant another node, which lies between the future left neighbor and right neighbor of node j, which will be initiated after the node j receives the “JoinReply” message.*

$$
\begin{aligned}
&\mathit{JoinGrantL} \triangleq \forall m \in \mathit{receivedMsgs}, g, i, j \in I : \\
&\quad \land\ m.mreq.type = \text{“JoinReply”} \\
&\quad \land\ m.mreq.lset.node = i \\
&\quad \land\ m.destination = j \\
&\quad \land\ g \in \mathit{grant}[i] \\
&\quad \land\ g \neq i \\
&\quad \land\ \mathit{LeftNeighbor}(\mathit{AddToLSet}(\mathit{GetLSetContent}(m.mreq.lset), \mathit{EmptyLS}(j))) = i \\
&\quad \Rightarrow \mathit{CwDist}(i, \mathit{RightNeighbor}(m.mreq.lset)) \leq \mathit{CwDist}(i, g)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{JoinGrantR} \triangleq \forall m \in \mathit{receivedMsgs}, g \in I : \\
&\quad \land\ m.mreq.type = \text{“JoinReply”} \\
&\quad \land\ m.mreq.lset.node = i \\
&\quad \land\ m.destination = j \\
&\quad \land\ g \in \mathit{grant}[i] \\
&\quad \land\ g \neq i \\
&\quad \land\ \mathit{RightNeighbor}(\mathit{AddToLSet}(\mathit{GetLSetContent}(m.mreq.lset), \mathit{EmptyLS}(j))) = i \\
&\quad \Rightarrow \mathit{CwDist}(\mathit{LeftNeighbor}(m.mreq.lset), i) \leq \mathit{CwDist}(g, i)
\end{aligned}
$$

Invariant 6.5.17. *As long as there is a “JoinReply” message from node i to node j , then node i must have the node j as its direct neighbor. Here the ls is introduced only*

to improve the readability of the invariant.

$$
\begin{aligned}
&JoinNeighborR \triangleq \forall m \in receivedMsgs, ls \in LSet, i, j \in I : \\
&\quad \land\ m.mreq.type = \text{“JoinReply”} \\
&\quad \land\ m.mreq.lset.node = i \\
&\quad \land\ m.destination = j \\
&\quad \land\ ls = AddToLSet(GetLSetContent(m.mreq.lset), EmptyLS(j)) \\
&\quad \land\ RightNeighbor(ls) = i \\
&\quad \Rightarrow LeftNeighbor(lset[i]) = j
\end{aligned}
$$

$$
\begin{aligned}
&JoinNeighborL \triangleq \forall m \in receivedMsgs, ls \in LSet, i, j \in I : \\
&\quad \land\ m.mreq.type = \text{“JoinReply”} \\
&\quad \land\ m.mreq.lset.node = i \\
&\quad \land\ m.destination = j \\
&\quad \land\ ls = AddToLSet(GetLSetContent(m.mreq.lset), EmptyLS(j)) \\
&\quad \land\ LeftNeighbor(ls) = i \\
&\quad \Rightarrow RightNeighbor(lset[i]) = j
\end{aligned}
$$

Invariant 6.5.18. *As long as there is a “JoinReply” message from node i, whose current left and right neighbors are identical, then this message must contain empty leaf sets. This situation refers to the situation when there is only one “ready” node in the network and some other node wants to join.*

$$
\begin{aligned}
&JoinSingleNode \triangleq \forall m \in receivedMsgs, i \in I : \\
&\quad \land\ m.mreq.type = \text{“JoinReply”} \\
&\quad \land\ m.mreq.lset.node = i \\
&\quad \land\ LeftNeighbor(lset[i]) = RightNeighbor(lset[i]) \\
&\quad \Rightarrow m.mreq.lset = EmptyLS(i)
\end{aligned}
$$

Invariant 6.5.19. *As soon as a “LeaseReply” message has been sent out from a node i to the node j knowing that node i has joined through j, then node i must be “ready”.*

$$
\begin{aligned}
&SemLReplyReady \triangleq \forall m \in receivedMsgs, i, j \in I : \\
&\quad \land\ m.mreq.type = \text{“LeaseReply”} \\
&\quad \land\ m.mreq.lset.node = i \\
&\quad \land\ m.destination = j \\
&\quad \land\ toj[j] = i \\
&\quad \Rightarrow i \in ReadyNodes
\end{aligned}
$$

Invariant 6.5.20. *The leaf sets of the sender and destination of any “LeaseReply” message are never empty.*

$$
\begin{aligned}
&SemLReplynonEmpty \triangleq \forall m \in receivedMsgs : \\
&\quad m.mreq.type = \text{“LeaseReply”} \\
&\quad \Rightarrow \land\ lset[m.mreq.lset.node] \neq EmptyLS(m.mreq.lset.node) \\
&\quad\quad\ \land\ lset[m.destination] \neq EmptyLS(m.destination)
\end{aligned}
$$

Invariant 6.5.21. *For any "LeaseRequest" message from node i to node j, we know that (1) j has not joined through i; (2) i must be either "ready" or "ok"; (3) neither i nor j has empty leaf sets; (4) j is farther to i than i's direct neighbors.*

$$
\begin{aligned}
& \mathit{SemLReq} \triangleq \forall m \in \mathit{receivedMsgs}, i, j \in I : \\
& \quad \land\ m.\mathit{mreq.type} = \text{“}\mathit{LeaseRequest}\text{”} \\
& \quad \land\ m.\mathit{lset.node} = i \\
& \quad \land\ m.\mathit{destination} = j \\
& \quad \Rightarrow \land\ \mathit{toj}[i] \neq j \\
& \qquad \land\ i \in \mathit{ReadyOK} \\
& \qquad \land\ i \neq j \\
& \qquad \land\ \mathit{lset}[i] \neq \mathit{EmptyLS}(i) \\
& \qquad \land\ \mathit{lset}[j] \neq \mathit{EmptyLS}(j) \\
& \qquad \land\ \mathit{CwDist}(i, \mathit{RightNeighbor}(\mathit{lset}[i])) \leq \mathit{CwDist}(i, j) \\
& \qquad \land\ \mathit{CwDist}(\mathit{LeftNeighbor}(\mathit{lset}[i]), i) \leq \mathit{CwDist}(j, i)
\end{aligned}
$$

Invariant 6.5.22. *If a node n has sent a "LeaseReply" message to i, we know that (1) n must be either "ready" or "ok"; (2) i is different from n; (3) if the granting bit is set to* TRUE *then n must have granted its lease to i ($i \in grant[n]$).*

$$
\begin{aligned}
& \mathit{SemLReply} \triangleq \forall i, n \in I, m \in \mathit{receivedMsgs} : \\
& \quad \land\ m.\mathit{mreq.type} = \text{“}\mathit{LeaseReply}\text{”} \\
& \quad \land\ m.\mathit{mreq.lset.node} = n \\
& \quad \land\ m.\mathit{destination} = i \\
& \quad \Rightarrow \land\ n \in \mathit{ReadyOK} \\
& \qquad \land\ n \neq i \\
& \qquad \land\ (m.\mathit{mreq.grant} = \text{TRUE} \Rightarrow i \in \mathit{grant}[n])
\end{aligned}
$$

Invariant 6.5.23. *For any "JoinReply" message, we know (1) its destination must still have empty leaf sets; (2) its receiver is joined through its sender; (3) the sender must be a "ready" node; (4) the receiver must be a "waiting" node.*

$$
\begin{aligned}
& \mathit{SemJRply} \triangleq \forall m \in \mathit{receivedMsgs} : m.\mathit{mreq.type} = \text{“}\mathit{JoinReply}\text{”} \\
& \Rightarrow \land\ \mathit{lset}[m.\mathit{destination}] = \mathit{EmptyLS}(m.\mathit{destination}) \\
& \quad \land\ \mathit{toj}[m.\mathit{mreq.lset.node}] = m.\mathit{destination} \\
& \quad \land\ m.\mathit{mreq.lset.node} \in \mathit{ReadyNodes} \\
& \quad \land\ m.\mathit{destination} \in \mathit{WaitNodes}
\end{aligned}
$$

Invariant 6.5.24. *As long as there is a "JoinReply" message, the current left neighbor and right neighbor of the sender cannot be farther from the sender than the neighbors according to the leaf set contained in the message.*

$$
\begin{aligned}
& \mathit{SemJoinLeafSet} \triangleq \forall m \in \mathit{receivedMsgs}, n \in I : \\
& \quad \land\ m.\mathit{mreq.type} = \text{“}\mathit{JoinReply}\text{”} \\
& \quad \land\ m.\mathit{mreq.lset.node} = n \\
& \Rightarrow \land\ \mathit{CwDist}(\mathit{LeftNeighbor}(\mathit{lset}[n]), n) \leq \mathit{CwDist}(\mathit{LeftNeighbor}(m.\mathit{mreq.lset}), n) \\
& \quad \land\ \mathit{CwDist}(n, \mathit{RightNeighbor}(\mathit{lset}[n])) \leq \mathit{CwDist}(n, \mathit{RightNeighbor}(m.\mathit{mreq.lset}))
\end{aligned}
$$

Invariant 6.5.25. *All the leaf set members contained within a "JoinReply" message must have no empty leaf sets themselves. The leaf sets of those nodes that are leaf set members contained within a "JoinReply" message are never empty.*

$$\begin{aligned}&\mathit{SemJRplyLSnonEmpty} \triangleq \forall m \in \mathit{receivedMsgs} : \\ &\quad m.\mathit{mreq.type} = \text{"JoinReply"} \\ &\quad\quad \Rightarrow \forall k \in \mathit{GetLSetContent}(m.\mathit{mreq.lset}) : \\ &\quad\quad\quad \mathit{lset}[k] \neq \mathit{EmptyLS}(k)\end{aligned}$$

Invariant 6.5.26. *If a node i is in another node j's leaf sets, and node i is not currently joining the network through j, then i's leaf sets are not empty.*

$$\begin{aligned}&\mathit{LSnonEmpty} \triangleq \forall i, j \in I : \\ &\quad \land\ i \in \mathit{GetLSetContent}(\mathit{lset}[j]) \setminus \{\mathit{toj}[j]\} \\ &\quad \land\ i \neq j \\ &\quad \Rightarrow \mathit{lset}[i] \neq \mathit{EmptyLS}(i)\end{aligned}$$

Invariant 6.5.27. *There are no two different "JoinReply" messages sent to the same node.*

$$\begin{aligned}&\mathit{OneJRply} \triangleq \forall m, mm \in \mathit{receivedMsgs} : \\ &\quad \land\ m.\mathit{mreq.type} = \text{"JoinReply"} \\ &\quad \land\ mm.\mathit{mreq.type} = \text{"JoinReply"} \\ &\quad \land\ m.\mathit{destination} = mm.\mathit{destination} \\ &\quad \Rightarrow m = mm\end{aligned}$$

Invariant 6.5.28. *If a node i owns the lease from n, then n must have granted its lease to i.*

$$\mathit{LeaseGrant} \triangleq \forall i, n \in I : n \in \mathit{lease}[i] \Rightarrow i \in \mathit{grant}[n]$$

Invariant 6.5.29. *As long as there is a "JoinRequest" message from a node i, the node must be still "waiting" and has no leaf set members.*

$$\begin{aligned}&\mathit{SemJReq} \triangleq \forall m \in \mathit{receivedMsgs} : m.\mathit{mreq.type} = \text{"JoinRequest"} \\ &\quad \Rightarrow \land\ m.\mathit{mreq.node} \in \mathit{WaitNodes} \\ &\quad\quad\ \land\ \mathit{lset}[m.\mathit{mreq.node}] = \mathit{EmptyLS}(m.\mathit{mreq.node})\end{aligned}$$

Invariant 6.5.30. *If a node i receives "JoinReply" message from r, then either its left neighbor becomes r and its right neighbor becomes the right neighbor of r or its right neighbor becomes r and its left neighbor becomes the left neighbor of r.*

$$\begin{aligned}&\mathit{SemJoinNeighbor} \triangleq \forall ls \in \mathit{LSet}, m \in \mathit{receivedMsgs} : \\ &\quad \land\ m.\mathit{mreq.type} = \text{"JoinReply"} \\ &\quad \land\ ls = \mathit{AddToLSet}(\mathit{GetLSetContent}(m.\mathit{mreq.lset}), \mathit{EmptyLS}(m.\mathit{destination})) \\ &\quad \Rightarrow \lor \land\ \mathit{RightNeighbor}(ls) = m.\mathit{mreq.lset.node} \\ &\quad\quad\quad \land\ \mathit{LeftNeighbor}(ls) = \mathit{LeftNeighbor}(m.\mathit{mreq.lset}) \\ &\quad\quad \lor \land\ \mathit{LeftNeighbor}(ls) = m.\mathit{mreq.lset.node} \\ &\quad\quad\quad \land\ \mathit{RightNeighbor}(ls) = \mathit{RightNeighbor}(m.\mathit{mreq.lset})\end{aligned}$$

Invariant 6.5.31. *The leaf sets of the sender of a "Probe" or "ProbeReply" message must not be empty. The node will never send "Probe" or "ProbeReply" message to itself or to the set of nodes it propagate to other nodes, which it believed to be faulty.*

$$
\begin{aligned}
& \mathit{SemProbePRply} \triangleq \forall m \in \mathit{receivedMsgs} : \\
& \quad \lor\ m.mreq.type = \text{“ProbeReply”} \\
& \quad \lor\ m.mreq.type = \text{“Probe”} \\
& \quad \Rightarrow \land\ lset[m.mreq.node] \neq EmptyLS(m.mreq.node) \\
& \qquad \land\ m.mreq.node = m.mreq.lset.node \\
& \qquad \land\ m.mreq.node \neq m.destination \\
& \qquad \land\ m.destination \notin m.mreq.failed
\end{aligned}
$$

Invariant 6.5.32. *The leaf sets of the destination of a "ProbeReply" message must be not empty. A node only receives "ProbeReply" from another node when it has probed that node.*

$$
\begin{aligned}
& \mathit{SemPRply} \triangleq \forall m \in \mathit{receivedMsgs} : \\
& \quad m.mreq.type = \text{“ProbeReply”} \\
& \quad \Rightarrow \land\ lset[m.destination] \neq EmptyLS(m.destination) \\
& \qquad \land\ m.mreq.node \in probing[m.destination]
\end{aligned}
$$

Invariant 6.5.33. *The leaf sets of a "dead" node are always empty.*

$$
\mathit{SemDead} \triangleq \forall k \in I : (status[k] = \text{“dead”}) \Rightarrow (lset[k] = EmptyLS(k))
$$

Invariant 6.5.34. *The sender of any "Probe" message must be awaiting a reply from the receiver by having the receiver in its probing set.*

$$
\begin{aligned}
& \mathit{SemProbe} \triangleq \forall m \in \mathit{receivedMsgs} : \\
& \quad m.mreq.type = \text{“Probe”} \\
& \quad \Rightarrow m.destination \in probing[m.mreq.node]
\end{aligned}
$$

Invariant 6.5.35. *The message "ProbeReply" from node i to j is the reply to message "Probe" from j to i. Therefore, both message can not exist at the same time.*

$$
\begin{aligned}
& \mathit{PrbMsgDual} \triangleq \forall mp, mr \in \mathit{receivedMsgs} : \\
& \quad \land\ mr.mreq.type = \text{“ProbeReply”} \\
& \quad \land\ mp.mreq.type = \text{“Probe”} \\
& \quad \Rightarrow \lor\ mp.mreq.node \neq mr.destination \\
& \qquad \lor\ mr.mreq.node \neq mp.destination
\end{aligned}
$$

Invariant 6.5.36. *There are no two different "ProbeReply" messages from the same sender and to the same receiver.*

$$
\begin{aligned}
& \mathit{OnePRply} \triangleq \forall m1, m2 \in \mathit{receivedMsgs} : \\
& \quad \land\ m1.mreq.type = \text{“ProbeReply”} \\
& \quad \land\ m2.mreq.type = \text{“ProbeReply”} \\
& \quad \land\ m1.mreq.node = m2.mreq.node \\
& \quad \land\ m1.destination = m2.destination \\
& \quad \Rightarrow m1 = m2
\end{aligned}
$$

Invariant 6.5.37. *There are no two different "Probe" messages from the same sender and to the same receiver.*

$$\begin{aligned}
&\mathit{OneProbe} \triangleq \forall m1, m2 \in \mathit{receivedMsgs} : \\
&\quad \land\ m1.\mathit{mreq.type} = \text{“}\mathit{Probe}\text{”} \\
&\quad \land\ m2.\mathit{mreq.type} = \text{“}\mathit{Probe}\text{”} \\
&\quad \land\ m1.\mathit{mreq.node} = m2.\mathit{mreq.node} \\
&\quad \land\ m1.\mathit{destination} = m2.\mathit{destination} \\
&\quad \Rightarrow m1 = m2
\end{aligned}$$

Invariant 6.5.38. *There are no two different "JoinRequest" messages from the same node.*

$$\begin{aligned}
&\mathit{OneJReq} \triangleq \forall m1, m2 \in \mathit{receivedMsgs} : \\
&\quad \land\ m1.\mathit{mreq.type} = \text{“}\mathit{JoinRequest}\text{”} \\
&\quad \land\ m2.\mathit{mreq.type} = \text{“}\mathit{JoinRequest}\text{”} \\
&\quad \land\ m1.\mathit{mreq.node} = m2.\mathit{mreq.node} \\
&\quad \Rightarrow m1 = m2
\end{aligned}$$

Invariant 6.5.39. *As a "JoinReply" comes to the message pool with the destination to i, there must be no "JoinRequest" message sent by i.*

$$\begin{aligned}
&\mathit{JoinMsgDual} \triangleq \forall m, mm \in \mathit{receivedMsgs} : \\
&\quad \land\ m.\mathit{mreq.type} = \text{“}\mathit{JoinReply}\text{”} \\
&\quad \land\ mm.\mathit{mreq.type} = \text{“}\mathit{JoinRequest}\text{”} \\
&\quad \Rightarrow mm.\mathit{mreq.node} \neq m.\mathit{destination}
\end{aligned}$$

Invariant 6.5.40. *As long as there is a "JoinReply" message containing empty leaf set in the message, then the sender of the "JoinReply" message must be the only "ready" node of the network.*

$$\begin{aligned}
&\mathit{SemSingleReadyJoin} \triangleq \forall m \in \mathit{receivedMsgs} : \\
&\quad \land\ m.\mathit{mreq.type} = \text{“}\mathit{JoinReply}\text{”} \\
&\quad \land\ m.\mathit{mreq.lset} = \mathit{EmptyLS}(m.\mathit{mreq.lset.node}) \\
&\quad \Rightarrow \mathit{ReadyNodes} = \{m.\mathit{mreq.lset.node}\}
\end{aligned}$$

Invariant 6.5.41. *In a "JoinReply" message, if either of the neighbor of the node according to the contained leaf set is the node itself, then the leaf set must be empty and vise versa. This is in fact a result of turning Lemma 4.1.3 to an invariant.*

$$\begin{aligned}
&\mathit{SemEmptyLSetSingleNode} \triangleq \forall m \in \mathit{receivedMsgs} : \\
&\quad m.\mathit{mreq.type} = \text{“}\mathit{JoinReply}\text{”} \\
&\quad \Rightarrow ((m.\mathit{mreq.lset.node} = \mathit{LeftNeighbor}(m.\mathit{mreq.lset}) \\
&\qquad \lor\ m.\mathit{mreq.lset.node} = \mathit{RightNeighbor}(m.\mathit{mreq.lset})) \\
&\qquad \Leftrightarrow m.\mathit{mreq.lset} = \mathit{EmptyLS}(m.\mathit{mreq.lset.node}))
\end{aligned}$$

Invariant 6.5.42. *When there is only one "ready" node in the network and there is some other node with non-empty leaf set, then this latter node should be assigned as joining node by the "ready" node.*

$$
\begin{aligned}
&SingleReadyNodeToj \triangleq \forall k \in I, r \in I : \\
&\quad \wedge\ ReadyNodes = \{r\} \\
&\quad \wedge\ lset[k] \neq EmptyLS(k) \\
&\quad \wedge\ r \neq k \\
&\quad \Rightarrow toj[r] = k
\end{aligned}
$$

Invariant 6.5.43. *There is at least one "ready" node on the ring. This is the invariant corresponding to the assumption that initially there is one "ready" node and no node leaves the network.*

$$AtleastOneReady \triangleq ReadyNodes \neq \{\}$$

6.5.2 Proof Sketch of the Invariant *IRN* as an Example

The invariant *IRN* states that no "ready" node r can be closer to a node i than its direct right neighbor.

$$
\begin{aligned}
&\forall i \in I, r \in \text{ReadyNodes} : i \neq r \\
&\quad \Rightarrow CwDist(i, RightNeighbor(lset[i])) \leq CwDist(i, r)
\end{aligned}
$$

Since this is in fact a strengthened invariant of property *NeighborClosest*, which is similar to the invariant *CompleteLeafSet* as illustrated in Invariant 6.3.3, the proof of the invariant *IRN* uses a similar approach as the one used in the proof for *CompleteLeafSet* explained in Section 6.3.3. The change of the two variables *lset* and *status* is critical.

With respect to the change of leaf sets, the proof uses Lemma 6.3.11 to show that adding nodes into leaf sets preserves the validity of the invariant and since no action in the new Pastry model removes nodes from leaf sets, the changes of leaf sets always preserve the invariant *IRN*.

Regarding the changes of *status* as in the proof of the invariant *CompleteLeafSet* it is enough to consider the action RecLReply(r) as specified in Definition 4.2.15 and to hypothetically construct a violation which is then to be refuted.

The node r only turns to "ready" when it is "ok" and fulfills the condition of *okToReady* in the action RecLReply(r). Applying *TojNoReady* in Invariant 6.5.12 on the "ok" node r, it can be concluded that there exists a node r_2, such that $toj[r_2] = r$. Applying *SemToj* in Invariant 6.5.10 on this result, node i is either the right or the left neighbor of node $r2$. This intermediate result will be used later (*).

The proof method now is to make a case analysis on the position of this node $r2$ as illustrated in Figure 6.8. According to *IRN*, either $r2$ is node i or $r2$ is outside the range from i to its right neighbor $rn(i)$, because otherwise *IRN* should have been violated.

Since node $r2$ must be "ready" to help joining node and according to *IRN*, node $r2$ cannot be inside the range from i to its right neighbor. Hence, the following 3 cases are possible for the position of node $r2$:

CASE 1 : $r2 = i$. By (*) and case analysis on the position of node $r2$ and its neighbor, this case that $r2 = i$ is impossible: If r were the right neighbor of $r2$, which is now i, then r should be the right neighbor of i, which contradicts with the assumption shown in Figure 6.8. If r were the left neighbor of $r2$, then the left distance (counter clockwise) from i to its left neighbor r is larger than the left distance from i to its right neighbor, which contradicts the definition of *LeftNeighbor* and *RightNeighbor* in Definition 4.1.2. Hence $r2$ cannot be i.

CASE 2 : $CwDist(i, RightNeighbor(lset[i])) < CwDist(i, r2)$. Using (*) again, we make case analysis on the node r, which is either left or right neighbor of $r2$. Suppose r is the right neighbor of node $r2$ as illustrated in Figure 6.8 and the other case can be proved symmetrically. The next step is to analyze the status of node i. If node i were a "ready" node, then this would violate *IRN* because then a "ready" node i would exist between a node $r2$ and its right neighbor r. Hence this case is impossible. If node i were not "ready" node, using invariant *TojNoReady* (Invariant 6.5.9) to construct an arbitrary positioned "ready" node $r4$, through which node i is currently joining. Then use the invariant *TojClosestL* (Invariant 6.5.8), which states that if node r is joined through some node $r2$, then between these two nodes, there exists no further node such as i, which is currently joining through another node $r4$. Hence, this case is impossible. Other cases are symmetrical to the cases analyzed here, therefore can be closed using the symmetrical invariants *TojClosestR* and *NRI* respectively.

CASE 3 : $r2 = RightNeighbor(lset[i])$. Here, we make a case analysis on the position of $ln = LeftNeighbor(lset[r])$, in order to show that such ln does not exist. That means, assuming such ln exists, contradiction must be derived. The node ln cannot be the same node as i, because according to *GrantNeighborNew* (Invariant 6.5.3), if node ln had granted the node r, then r could not be closer than its right neighbor $RightNeighbor(lset[i])$. But due to the assumption that node r is "ok" and is becoming "ready", it must already get the grant from its left neighbor ln. Hence contradiction is derived, such that this case is impossible.

The node ln can not be left to i as illustrated in $ln2$ in CASE 3 of Figure 6.8. In this situation, node i cannot be "ready" because it locates between a node r and its left neighbor ln. According to *TojNoReady* (Invariant 6.5.9), for the node i that is not "ready", there exists a node $r3$, through which node i is currently joining. Then the left task is to refute the existence of such a node $r3$. The node $r3$ cannot be $r2$, hence, node $r3$ must be the left neighbor of i. According to *TojDualR* (Invariant 6.5.11) together with *SemToj* (Invariant 6.5.10), node i must be the right neighbor of $r3$. The next step is to discuss the position of the $r3$ as the left neighbor of i. On the one hand, it must exist between i and ln, because if node ln is "ready", it can not lie between a node i and its left neighbor $r3$ by *NRI*. But on the other hand, node $r3$ cannot lie between i and ln, because $r3$ is ready and it should not lie between a node r and its left neighbor ln. Therefore, $r3$ can only be equal to node ln. Thus the precondition for *GrantHistL* (Invariant 6.5.4) is fulfilled, which applies here with the following statement: If a not yet "ready" node i lies between two other different nodes $r3$ (as well as ln) and r, and node i is joined through one of the node, i.e. node $r3$, whereas this node, again node $r3$, has granted its lease to the other node r (according to the assumption that node r

is turning from "ok" to "ready"), then the direct neighbor of i, which is $r2$, must be closer to i than the other node r. Regarding the last case in Figure 6.8, $r2$ is not closer to i than r. Hence a contradiction is derived, such that node *ln* cannot be left to i as shown in CASE 3 of Figure 6.8.

Now regard how *ln* cannot exist between i and r. By the inductive hypothesis of *IRN*, *ln* cannot be "ready", because it lies between a node i and its right neighbor $r2$. Then again by *TojNoReady* (Invariant 6.5.9), there exists a node $r5$, such that $toj[r5] = LeftNeighbor(lset[r])$. The next step is to make a case analysis of the position of $r5$. Because of *IRN*, it cannot be inside the range $[i, rn(i)]$. Because of *TojClosestL*, $r5$ cannot be outside the range of (i, r). Hence, $r5$ cannot exist. Hence, node *ln* cannot lie between i and r.

In conclusion, there is no possible position for such a node *ln* to exist, which means that there exists no node to grant node r its lease to make it "ready", and therefore, the constructed violation is impossible. Thus the proof is competed.

6.6 The TLA$^+$ Proof

The proof of the inductive invariant *InvLuPastry* in TLA$^+$ is constructed hierarchically. The induction step is shown in the example in Figure 6.9. The TLA$^+$ proof language illustrated here is introduced in Section 2.3.6.

The head of the proof (first two lines in Figure 6.9) claims the induction step as a lemma with name *InvInvariant*. It is a sequent assuming that the induction hypothesis *InvLuPastry* holds and concluding that the post condition *InvLuPastry′* still holds. The type correctness *TypeInvariant* is inductively proved as a separate theorem, which is used as an assumption here.

The following lines are the proof body of the lemma *InvInvariant*, which consists of several steps as case analysis, each of which targets a particular action from *Next* or the stuttering step UNCHANGED *vars* for the case no action is executed. Each branch of the proof is finalized with a QED step. For example, the last step ⟨1⟩99 finalizes the proof with a QED step by summarizing all the cases, which confirm the definition of *Next*.

The arguments for a proof is always introduced with the keyword BY with the used lemmas, either with their names or with the step number within the local proof as shown in the last step ⟨1⟩99. Here in step ⟨1⟩99, the step number ⟨1⟩1, ⟨1⟩2 and ⟨1⟩98 are the previously proved lemmas, each representing that the *InvLuPastry* holds after execution of a particular action.

In the proof arguments of step ⟨1⟩99, the keyword *SMT* stands for a proof method used by the TLAPS prover system, telling the TLAPS prove manager to launch the *SMT* back-end prover to solve this step of proof.

Take a particular proof step ⟨1⟩1 as an example to see how a sub-proof is manually conducted in TLA$^+$, if a simple BY ... DEF ... arguments can not be proved by the back-end prover. This step states a sequent that asserts that *InvLuPastry* is preserved by the action Deliver(i, j) for any $i, j \in I$, guided by the structure of formula Next. The back-end prover is not able to find a proof in a huge search space when all the

definitions are unfolded and all the proved lemmas are used at once. Therefore, using divide and conquer techniques to unfold only the necessary definition and to include only the needed lemmas for local proof is the most challenging task left to the proof writer. It is said that by the time this thesis is written, the Toolbox (GUI) of TLA$^+$ helps in obvious decompositions by suggesting the high-level structure of the proof.

For example, the sub-proof step $\langle 2 \rangle 1$ restates the proof obligation of its parent level $\langle 1 \rangle 1$. by further unfolding the definition of the action Deliver(i, j). Here the enabling condition of Deliver(i, j) as shown in Definition 4.2.2 is not included in the set of assumptions listed in the step $\langle 2 \rangle 1$, because for this particular proof obligation *NeighborClosest*, this information is useless.

The assumptions introduced in the ASSUME part are always the ingredients for the following proof steps, which can be referred by its step number. For example the sub step $\langle 3 \rangle 9$ uses some statement from $\langle 2 \rangle 1$ by referring to the step number.

The conjuncts of *InvLuPastry* are proved one by one after execution of each action. As illustrated in Figure 6.9, the sub-proof step $\langle 2 \rangle 25.\mathit{GrantOK}'$ shows that the invariant *GrantOK* holds after execution of action Deliver(i, j), as mentioned in the parent proof step $\langle 1 \rangle 1$. The step $\langle 2 \rangle 99$ of each sub-proof for a particular action finalizes the local proof by summarizing all the sub-proofs of the invariants.

In some proofs, an additional proof step is invented as a local lemma, which is frequently referred in the later local proof steps. Take the sub-proof step $\langle 2 \rangle 2$ of proof step $\langle 1 \rangle 2$ as an example which is the easiest case of such kinds of lemmas. Here the arguments for proving this lemma includes many external lemmas, such as *StatusDisjoint* and *ReadyInI*.

These lemmas are separately proved as preparation for the final inductive proof. Some of them are lemmas of types, such as *ReadyInI* states that all the "ready" nodes are also elements of I; others are lemmas of properties of the data structure used for state variables, such as *StatusDisjoint* states that the status of a node is unique, such that a "ready" node cannot be at the same time a "waiting" node. Here using the definition of invariant *SemJReq*, which states that a node sending a "JoinRequest" message must be a "waiting" node, and the lemma *StatusDisjoint*, it can be inferred that the sender of the received "JoinRequest" message $m.mreq.node$ cannot be the same node as i because node i is a "ready" node according to the assumption listed in step $\langle 2 \rangle 1$.

The final TLA$^+$ proof for the inductive invariant consists of more than 14500 lines without any comments as the lines illustrated in Figure 6.9. This does not include the repeating sub-proof steps, which are reformulated as external lemmas in a separate proof file consisting 600 lines. Additionally, the type correctness is also proved inductively in about 1000 lines. The lemmas of data structures such as *StatusDisjoint* and further lemmas about the general ring structure of Pastry are proved separately in about 2000 lines. These proofs with more than 20,000 lines, corresponding to more than 10000 proof steps, are all automatically verified using TLAPS proof manager, which launches different back-end first-order theorem provers or an extension of ISABELLE to find the proof and the proof can partially be checked by the core of ISABELLE.

Here more than 200 proof steps of the proof on invariants launch SMT solver. These steps simplify about 5 proof steps that consist of only explicit instantiation of the defini-

tions of invariants and some simple simplification proof steps into one proof step, which is not feasible to be automatically proved by other back-ends such as ZENON and ISABELLE/TLA. However, it is not yet possible to check these proofs from SMT solver in TLAPS using the core of ISABELLE. When the complete proof sketch of inductive invariants is iteratively verified, the proof can be checked automatically as a whole by one click in TLAPS, which at the end takes 4 hours using a Dell laptop with Intel Core i3-2330M CPU at 2.2GHz with full usage of 8GB memory.

There are still leftover proofs for the lemmas on the data structure, in particular the ring calculation which involves modulo arithmetics. Redlog[3] is used to verify most of them.

[3]http://redlog.dolzmann.de/

Definition 6.5.2 (*InvLuPastry*).

$\wedge$ *IRN* $\wedge$ *NRI*
$\wedge$ *GrantNeighborNew*
$\wedge$ *GrantHistR* $\wedge$ *GrantHistL*
$\wedge$ *GrantOK*
$\wedge$ *CoverNoNode*
$\wedge$ *CoverNoToj*
$\wedge$ *TojClosestL* $\wedge$ *TojClosestR*
$\wedge$ *TojNoReady*
$\wedge$ *SemToj*
$\wedge$ *TojDualR* $\wedge$ *TojDualL*
$\wedge$ *TojReady*
$\wedge$ *TojInjective*
$\wedge$ *LSnonEmptyToj*
$\wedge$ *JoinGrant*
$\wedge$ *JoinNC*
$\wedge$ *JoinSingleNode*
$\wedge$ *JoinNeighborR* $\wedge$ *JoinNeighborL*
$\wedge$ *SemLReplynonEmpty*
$\wedge$ *SemLReplyReady*
$\wedge$ *SemLReq*
$\wedge$ *SemLReply*
$\wedge$ *SemJRply*
$\wedge$ *SemJoinLeafSet*
$\wedge$ *SemJRplyLSnonEmpty*
$\wedge$ *LSnonEmpty*
$\wedge$ *OneJRply*
$\wedge$ *SemDead*
$\wedge$ *SemPRply*
$\wedge$ *SemProbe*
$\wedge$ *PrbMsgDual*
$\wedge$ *OnePRply*
$\wedge$ *OneProbe*
$\wedge$ *SemProbePRply*
$\wedge$ *HalfNeighborExt*
$\wedge$ *LeaseGrant*
$\wedge$ *SemJReq*
$\wedge$ *SemJoinNeighbor*
$\wedge$ *OneJReq*
$\wedge$ *JoinMsgDual*
$\wedge$ *SemSingleReadyJoin*
$\wedge$ *SemEmptyLSetSingleNode*
$\wedge$ *SingleReadyNodeToj*
$\wedge$ *AtleastOneReady*

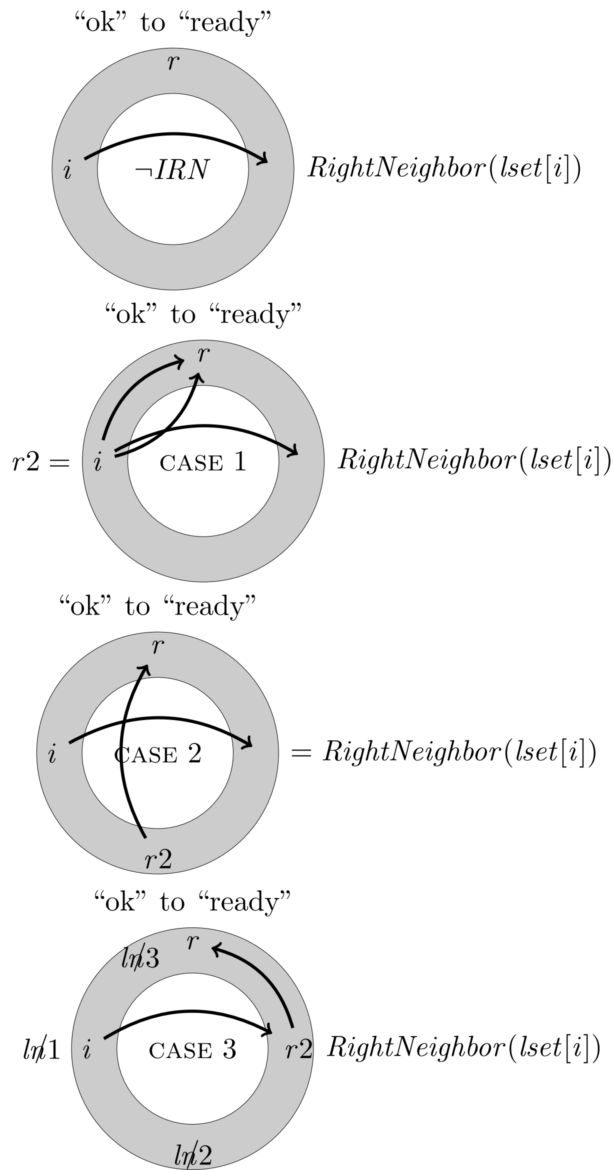

Figure 6.8: Hypothetical violation of *IRN* by its inductive proof.

```
LEMMA InvInvariant ≜ ASSUME TypeInvariant, Next_vars, Inv
                     PROVE Inv'
⟨1⟩1. ASSUME NEW i ∈ I, j ∈ I,
            Deliver(i, j), TypeInvariant, Inv
      PROVE Inv'
      ⟨2⟩1.SUFFICES ASSUME NEW m ∈ receivedMsgs,
          receivedMsgs' = receivedMsgs \ {m},
          UNCHANGED ⟨status, lset, rtable, probing, failed, lease, grant, toj⟩,
          TypeInvariant, Inv
          PROVE Inv'
        BY ⟨2⟩1, ⟨1⟩1DEF Deliver
      ⟨2⟩25.GrantOK'
        ⟨3⟩1.SUFFICES ASSUME ... PROVE ...
        ...
        ⟨3⟩9.QED BY ⟨3⟩1, ⟨2⟩1, ...
      ⟨2⟩26.LeaseGrant'
      ...
      ⟨2⟩99.QED BY ⟨2⟩25, ⟨2⟩26, ...
⟨1⟩2. ASSUME NEW i ∈ I,
            RecJReq(i), TypeInvariant, Inv
      PROVE Inv'
      ⟨2⟩1.SUFFICES ASSUME status[i] = "ready", ...
          PROVE Inv'
        BY ⟨2⟩1.⟨1⟩2.DEF RecJReq
      ⟨2⟩2.m.mreq.node ≠ i
        BY⟨2⟩1, StatusDisjoint, ReadyInI DEF ReadyNodes, SemJReq, Inv
      ⟨2⟩25.GrantOK'
      ...
      ⟨2⟩99.QED BY ⟨2⟩25, ...
...
⟨1⟩98. CASE UNCHANGED vars
         ...
⟨1⟩99. QED BY ⟨1⟩1, ⟨1⟩2, ..., ⟨1⟩98, SMT DEF Next
```

Figure 6.9: TLA$^+$ codes of the inductive proof of invariants.

7 Related Works

7.1 Other P2P Systems

As classified in Section 2.1.2, there are other types of *P2P* systems using different techniques and terminologies. The purpose of this section, however, is not to restrict the criteria for different categories, but to clarify the essence of the origins of different techniques and to provide a global rough picture of most popular *P2P* techniques. Figure 7.1 compares the virtual topologies of different distributed systems.

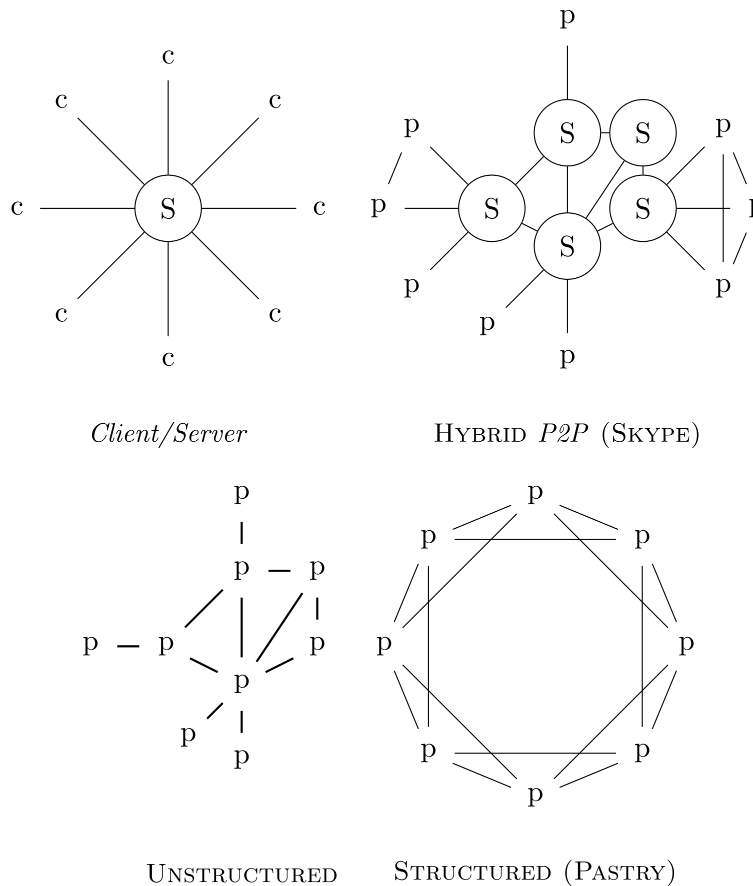

Figure 7.1: Different topologies of distributed systems.

In the following sections, they will be shortly introduced with their advantages and

disadvantages. A summary of comparison is shown then in Table 7.1.

7.1.1 Unstructured Decentralized Systems

Different from the structured *P2P* system as Pastry, unstructured *P2P* systems do not have particular constraints on the links between different peers. When a node joins the network, it typically initializes its links by repeatedly performing a *random walk* through the overlay starting at the *seed* and requesting a link to the node where the walk terminates. More random walks may be performed when the connection degree of a node falls below some minimum; they refuse link requests when the maximum degree is exceeded. Therefore, the minimum node degree is typically decided to maintain the connectivity against node failures and membership changes, whereas the maximum degree is set to bound the overhead maintaining the links.

Unstructured *P2P* systems store the content in the inserting node and the node which downloads the content. In addition, they also benefit from the random walk path for content storage. To obtain and locate the content, a node floods a request message, specifying a key, meta-data or keywords, through its connected nodes. If one of these nodes itself or some node in its routing table matches the search criterion, it responds to the query node. If not, the search continues by flooding the request two hops away from the query node, and so on, until some node can answer the query by returning the node address which holds the content. Alternatively, the query node can also use random walk as a search method instead of flooding as introduced in Rodrigues and Druschel (2010).

Coordination in the unstructured *P2P* system is by nature epidemic, i.e. information spreads like virus from one node to its neighbors. Therefore it is simple and robust. A more effective way is to form a spanning tree using a decentralized algorithm as mentioned in Rodrigues and Druschel (2010). This tree can then be used to multi-cast messages to all members or to compute summaries of the group. No matter which techniques are used, a balance needs to be kept between the efficiency and the overhead to maintain the system to be able to provide this efficiency. In the worst case, answering a query could lead to exploration of the complete network.

One of the successful techniques for unstructured *P2P* systems is the swarming protocol, used by BitTorrent and PPTV. Unlike the tree-based protocols introduced above for coordination and content distribution, swarming protocols do not have any structure. Content is divided into small bulks and multi-casted to all overlay nodes. In each swarming interval, overlay neighbors exchange knowledge indicating which blocks they have. They intersect the availability information received from their neighbors and request a block they do not have from one of its neighbors who has it. Fairness and randomness play here a key role.

Compared to *Client/Server* systems, unstructured decentralized *P2P* systems scale much better and are more resilient to failures, attacks and legal issues, due to the lack of a central controller as the target. The simplicity and robustness of unstructured systems make them quite widely adopted by non-profit communities such as Gnutella, and for sharing bulk data in trackless BitTorrent (the newer version). In fact, the trackless

BitTorrent uses an unstructured overlay for dissemination of data, but uses a key-based search for locating the resource provider. Key-based search is then a technique used in structured system, which will be introduced in a sequel.

7.1.2 Hybrid (Partly Centralized) P2P Systems

Hybrid *P2P* systems are a compromise between *Client/Server* systems and pure *P2P* systems. They always have a controller (or many super-nodes) that maintains a set of peers, and coordinate them to communicate among each other, or to find remote peers. For example, Napster maintains its membership and content index on a Web site. Skype also has a central site to provide utilities for the members to log-in and pay.

When new nodes join in a hybrid system, they first connect to the controller, which provides a set of nodes it manages and coordinates. Then peers set up the connections among themselves based on the information they get from the controller.

In addition to the controller, some hybrid systems have more complicated structure. For instance, Skype has developed the Global Index technology ("GI" or a so-called third generation of P2P technology "3G P2P") on a multi-tiered network where *Super Nodes* communicate in such a way that every such node in the network has full knowledge of all available users and resources with minimal latency. According to Goodin (2012), *Super Nodes* are those *P2P* nodes dedicated from the company side on servers within secure data centers. However, it is not clear if they run *P2P* protocol among themselves. "This has not changed the underlying nature of Skype's *P2P* architecture, in which *Super Nodes* simply allow users to find one another (calls do not pass through *Super Nodes*)", reported in Goodin (2012).

In Hybrid *P2P* systems, the typical places to store an object are the node that inserts the object and any nodes that subsequently download the object. Since the controllers maintain the information about which nodes exist in the system, as well as keys and other attributes of them, the queries for a given key or a set of keywords are answered by those controllers, which respond to the queries with a set of nodes from which the requested resources can be obtained. In the same manner, coordinations for streaming content or telephone channel can be set up directly through the central points (or coordinated through *Super Nodes*).

Despite central controllers or trackers, sharing of the content are carried out by the peers introduced through those central nodes, to release the throughput on the service initiator. After coordination is completed through the central nodes, the requesters set up *P2P* connections with the peer nodes, download the information and act by default like them to share the resources.

Based on global coordination and local sharing, Hybrid *P2P* systems can provide more organic growth and ample resources, and are more scalable and resilient to failures and attacks than pure *Client/Server* systems, but more restricted than those of pure *P2P* systems. The central nodes often form potential bottleneck and a single point of failure and attack, reported by Rodrigues and Druschel (2010). In contrast to pure *P2P* systems, the hybrid systems are often better to manage and control. Therefore they are widely adopted by enterprises such as Napster, PPTV, PPStream, Skype.

Classification	Hybrid	Unstructured	Structured
Examples	NAPSTER, SKYPE, BITTORRENT	GNUTELLA, KAZZA, trackless BITTORRENT	Pastry, Chord
Techniques/ Terminology	tracker, *Super Nodes*	seed, random walk, flood, swarming, spanning tree	*DHT*, shared-prefix, logarithmic hop steps
Content Distribution	inserting node, downloaders	swarming, along random walk path	set of neighbors, pointers
Search& Coordination	controller, keywords-based, central index	no index, flood, keywords-based,	index in *DHT*, key-based routing, spanning tree
Pros	easier management, better control	scale very well, simple and robust, resilience, complex keywords search	scale very well, efficient routing, less overhead, resilience
Cons	restricted scalability, vulnerable for single point failure	large routing cost more overhead, no control	complicated algorithm, suffer from churn, no keywords search

Table 7.1: Classification of *P2P* Systems

7.1.3 Summary of P2P Systems

Table 7.1 summarizes the mentioned points to the different categories discussed above and give a comparison of them. In the mean time, these techniques are adopted by each other in the real applications, such that a clear classification of a *P2P* system is no more possible. The message to take is that a practical *P2P* system cannot simultaneously achieve all three goals of scalability, availability and resilience to churn, a system can only choose two of them according to Blake and Rodrigues (2003).

7.2 Formal Analysis of Chord

Similar to Pastry, Chord (Stoica et al. (2001)) is also a virtual ring implementation of *DHT*. They share the ring topology and similar mechanisms for join, departure of nodes and their neighborhood maintenance. However, Chord has been described with a more formal specification, which makes it the target of many verification approaches.

A recent approach is Zave (2012), which uses Alloy to model Chord at a much higher level of abstraction than the model shown in this thesis for Pastry.

An important difference between Chord and Pastry is that Chord as described in Stoica et al. (2001) uses *predecessor* and *successor* according to their assignment in an action rather than a left (counter clockwise) and right (clockwise) neighbor of Pastry according

to their leaf sets. Therefore, Pastry has a correct local ordering by construction, that means, a left neighbor of a node is never numerically closer to a node than its right neighbor on the clockwise direction. While predecessor and successor of a node in Chord ring may become disordered when nodes concurrently join at one node.

Complementary to the drawback of Chord due to this difference, concurrent joins in Chord between two adjacent nodes on the ring can only disrupt the *successor*, because new node j is added by replacing j's hypothetical predecessor's successor by the pointer of j and assign the successor of j to be the previous successor of j's predecessor. A concurrent joining node between j's predecessor and successor can only join through its predecessor. On the opposite, Pastry suffers from neighborhood conflicts as illustrated in Section 3.2.1 because concurrent joins may disrupt both adjacent nodes, as a concurrent joining node of j between its hypothetical left neighbor (predecessor) and right neighbor (successor) can be joined through its right neighbor (successor).

7.2.1 Formal Analysis on Chord Using Alloy

Zave (2012) uses Alloy to formally analyze Chord on an abstract level. Many non-trivial counterexamples are discovered, which demonstrate the benefits of light-weight verification using model checking and model enumeration techniques of Alloy. The abstract model in Zave (2012) is compared with the formal specification in this thesis as follows.

By modeling the static structure, the concept of "between" corresponding to three nodes on ring is taken care of in both approaches. The difference is that the specification of *between* in Zave (2012) is on a higher abstract level. It invokes the boolean function $lt[n1, n3]$ to check if a node $n1$ is "less than" another node $n3$. But the semantics of "less than" is not clearly explained. This makes the correctness of *between* in Zave (2012) vulnerable to some intuitive refinement, such as implementing $lt[n1, n3]$ as $n1 \leq n3$ or $n1 < n3$. In this implementation, $between[0, 2^M - 2, 2^M - 1]$ will return FALSE, although it should be TRUE, since $2^M - 1$ is the largest identifier whose successor is 0. In contrast to Zave (2012), this thesis gives a detailed formal specification on the ordering of distances on a ring by comparing the *counter clockwise distances* as specified in Definition 4.1.1 in Section 4.1.1.

By verifying the dynamic behavior of the systems, both approaches provide a proof for the "pure join" cases, though with different interpretations of this concept. The proof in Zave (2012) relies on atomic and non-interfering executions of *join* and *stabilize* steps that update local neighborhood knowledge. As mentioned before, concurrent join in Chord only disrupt the successor, which leads to the result that pure join trivially does not violate the property *ValidSuccessorList* defined in Zave (2012), which states that a node should not ignore an existing node between itself and its successor. However, Pastry may easily violate the property *NeighborClosest*, which is similar to *ValidSuccessorList*, by concurrent join problem illustrated in Section 3.2.1.

Besides the similar actions like stabilization in Chord, the pure join model in this thesis allows all kinds of interleaving of *join*, lookup and route, which enlarges the scale of the problem.

One further difference is message interleaving. Since Zave (2012) assumes that

“nodes can read the states of adjacent nodes”, message exchange is cleverly abstracted away. But due to the nature of distributed systems, message delivering is rather essential because of possible disruptions caused by message interleaving. This thesis models the messages directly, in order to provide a standard framework of modeling distributed systems. In fact, allowing message interleaving does introduce a lot of unexpected challenges as illustrated in Section 5.2. More precisely, the variable *toj* is introduced to accomplish blocking of further joins that leads to the problem illustrated in Section 3.2.4. These problems are not discoverable on an abstraction level as Zave (2012).

Besides the comparison on the abstraction level, the desired property is also comparable. Zave (2012) makes efforts on analyzing connectivity and reachability of a network, i.e. if nodes can form a ring by their predecessor and successor and if all nodes are reachable/connected through some path on the ring, although the ordering of the nodes on the ring is also one of the invariants for proving these properties. In contrast to Zave (2012), this thesis focuses on collision-free coverage of the nodes (*CorrectDelivery* specified in Property 4.3.4), i.e. no “ready” node share the responsibility of keys with another node. This property requires more rigorous implementation of the protocol. Therefore, the corresponding set of invariants as shown in Definition 6.5.2 are therefore also more complicated than those proved in Zave (2012).

Additional perspective of comparison is the improvement of the protocol. Zave (2012) aims rather at finding the errors while this thesis aims at proposing a confident solution. Although a flaw was found in the original description of the algorithm of Chord, for which Zave (2012) suggests a fix that may be correct, there is no formal proof of this solution. This thesis suggests a series of improvements of the Pastry protocol in Section 3.2.4 and gives a formal description of a valid system in Chapter 4 with respect to desired properties and provides a formal proof of them in Chapter 6.

In fact, Alloy is not supported by a theorem prover like TLAPS to formally prove the invariants. Therefore, the formal analysis in Zave (2012) is based on model checking and model enumeration, which provides assertions only on a restricted number of nodes and leaves the generalization with an informal proof. Using TLAPS, this thesis is able to deliver a complete induction proof of the desired temporal property of the system and the proof is also automatically verified by TLAPS.

7.2.2 Previous Formal Approaches Analyzing Chord

To the best of our knowledge, no approaches exist besides this thesis for formal verification of Pastry. However, several approaches to analyzing Chord can be found in literature, such as Lynch and Stoica (2004), Li et al. (2004), Risson et al. (2005) and Bakhshi and Gurov (2007).

On a similar abstract level as this thesis, Lynch and Stoica (2004) provides formal description of the protocol as transition system considering concurrent join and leave and assumes no message loss. However, Lynch and Stoica (2004) focuses on providing fault tolerance by bounding the latency of messages and the number of leaving and concurrent joining nodes, while this thesis discusses and proves the guarantee of the safety property *CorrectDelivery*. Besides, this thesis provides a proof that is automatically checkable.

Li et al. (2004) models an abstract version of Chord with active departure protocol. Active departure and active join means that the node contacts the adjacent nodes before they join or leave the network. Li et al. (2004) argues that active departure is useful and more efficient than passive approaches. Li et al. (2004) provides also an "assertional proof" of pure join protocol of Chord.

Li et al. (2004) defers from this thesis firstly by its abstract level, where message interleaving is omitted, i.e. it implicitly assumes no message delay. However, this thesis assumes message delay by modeling non-deterministic execution of actions that allows message interleaving. The second difference lies in the verified property. This thesis focuses on correct key-addressing despite concurrent join, while Li et al. (2004) proves the connectivity of the Chord ring under join. The proof in Li et al. (2004) in a sketch way is quite convincing. However, this thesis has discovered several flaws of the proof sketch of LuPastry and used TLAPS to automatically check the final proof to have more confidence.

Further results from Li et al. (2004) include a conclusion that trivial combination of the leave protocol and the join protocol would cause live-lock, which then must be handled using dicing as proposed in Lehmann and Rabin (1981). Since this thesis has not provided a solution for node departure, the combination of active departure and the current join protocol in LuPastry can be an interesting future work to see if *Correct-Delivery* can be preserved.

In Risson et al. (2005), Event-B is applied to specify the Chord protocol with active departure. Implementing the Paxos commit protocol, which is already written in TLA$^+$, Risson et al. (2005) suggests a topology maintenance protocol providing the continuity of the ring with some fault-tolerance. It does not prove these properties but rather provides an analysis result involving 3 nodes on the ring and one faulty nodes, while in this thesis, model checking analysis and validation are carried out by 4 nodes and in a simplified version even with 5 nodes.

Bakhshi and Gurov (2007) also only considers join and assumes implicitly no departure of nodes. Different from the thesis, its proof shows that "the stabilization algorithm will eventually fix the immediate successor of each node" and the network will "eventually form a ring topology again". This thesis have proposed these analysis result on Pastry as a *validation* approach in Section 5.3, where a violation of an artificially formulated safety property with negative intention evidences the eventual success of join and lookup. Such kind of analysis from both approaches have the drawback that they do not provide any guarantee. For this reason, this thesis proposed further approach using theorem proving techniques to show the guarantee of safety properties such as *CorrectDelivery*.

7.3 Other Formal Methods Applied on Verifying Correctness of Network Protocol

Another formal method used for verification is process algebra. System is specified using algebraic structure and the verification is the algebraic calculation process to derive that

the refinement or implementation *simulates* the abstract representation. Since *DHTs* are implementations of abstract hash tables with a distributed implementation, they provide a good application case for using this approach.

Karsten et al. (2007) proposes an axiomatic basis for message delivery and addressing problems of network. It uses Hoare-style logic to specify the handling of messages at a network node and extend the logic with *leads-to* relation to mimic the store-and-forward principle.

A correctness proof for correct forwarding is established in Karsten et al. (2007) by symbolically reducing the post-condition to a trivially true pre-condition applying all possible operations.

Borgström et al. (2004) uses CCS (a typical process algebra language) to specify the abstract model of the hash table and its implementation with Distributed K-ary Search (DKS) consisting of request, route and lookup actions. It shows that these two models are weakly bi-simulated. This approach shares its interest with the thesis in verifying correctness of routing actions. But it ignores the dynamics of a *DHT* system and only focuses on the static case, where membership of the network does not change.

Bakhshi and Gurov (2007) first uses π-calculus to model Chord on an abstract level by employing an additional token ring protocol using the Chord and then shows that the refinement of the distributed algorithm used by Chord simulates the abstract protocol by showing their behavior equivalence.

Personally speaking, the proof and model in process algebra is far away from an executable implementation of a system, whereas the comparable results shown in this thesis using TLC is more intuitive for a software engineer to understand and to implement.

Besides papers, there are also extensive researches such as Ph.D. theses on relevant topics of this thesis.

The Ph.D. thesis Ghodsi (2006) has generalized the ring-based implementations of *DHT* to a DKS as already introduced in Borgström et al. (2004) and mentioned that Pastry, Chord can be instantiated from DKS. It discusses several common problems such as concurrent join, departures of nodes, separation of networks and introduces novel applications of *DHT* system in group communication, bulk operation and replication. Same as this thesis, Ghodsi (2006) also argues that consistency is impossible in the case of node failures due to the network partition problem, though no solution is proposed in either approach. Its point of view lies in extend the application of *DHT* only to *P2P* systems like SKYPE, where peers are unpredictable users with all kinds of abnormal behaviors, but also to ordinate clouds of servers, where departure of nodes is not common. Sharing this philosophy, the formal verification approach in the thesis can be implemented in such applications as well, provided with a trustful proof of correctness of LUPASTRY on the safety property *CorrectDelivery*.

However, Ghodsi (2006) differs from this thesis because it tries to use a lock to solve this problem of concurrent join on the abstract level. This solution was once considered during the development of IDEALPASTRY but then refuted for its unavailability, which leads to its improvement in LUPASTRY. In fact, as long as a node joins to the network in that model of Ghodsi (2006), at least three nodes are locked and there is no obvious way to unlock if a concurrent join at the neighbor node occurs before the join has

accomplished.

Ghodsi (2006) also shows that consistency can be preserved in the case of joining and voluntary leaving of nodes, though not as a safety property, but rather as the validation result similar to Bakhshi and Gurov (2007). Besides, Ghodsi (2006) seems not to use formal method to verify its claims, whereas this thesis has provided complete formal proof of IDEALPASTRY and LUPASTRY.

The Ph.D. thesis Bongiovanni (2012) provides an automatically checkable proof by ISABELLE for verifying the CAN *P2P* system, which implements *DHT* in a different way from Ring structures as Pastry and Chord.

In Bongiovanni (2012), the dynamic behaviors of the system are not modeled, i.e. the structure and membership of the network is assumed to be static, whereas this thesis investigates much efforts on verifying the correctness of the safety property under all possible joins of new nodes and provides a proof which is automatically checkable using TLAPS.

8 Conclusion and Future Work

The novelty and contributions of this thesis are summarized in the form of answers to the proposed research goals stated at the beginning of this thesis in Sections 1.1 and 1.2.

8.1 Answers to the Questions About Pastry

How does Pastry work, in particular, how does the protocol realize the DHT?

As explained in Section 3.1.1 in detail, Pastry is a structured *P2P* algorithm realizing a Distributed Hash Table (*DHT*) over an underlying virtual ring of nodes with logical identifiers from an ID space. The IDs serve two purposes: both as keys and identifiers of the nodes, such that an overlay node is in particular responsible for keys that are numerically close to its ID, i.e. it provides the primary storage for the hash table entries associated with these keys. Key responsibility is divided equally according to the distance between two neighbor nodes.

Pastry is supposed to provide "dependable routing" according to Castro et al. (2004), i.e. a lookup message for a key will be answered by the unique node which is responsible for that key. Nodes may join and leave the network dynamically, which is called *churn*. Therefore, Pastry maintains its correct key mapping in the presence of *churn* through its *join* protocol, which eventually adds a new node with an unused network ID to the ring.

This thesis formally modeled the protocols of Castro et al. (2004) as CastroPastry. Fine-grained pseudocode of CastroPastry is shown in Section 3.1.3. This model is further improved to HaeberlenPastry based on the formal analysis results from model checking CastroPastry using TLC and inspiration from Haeberlen et al. (2005) and FreePastry (2009). HaeberlenPastry requires an explicit "transfer of coverage" to a joining node from its neighbors before it can answer lookup requests. In the formal model HaeberlenPastry, joining nodes are assigned different statuses from "dead" to "ready". Only "ready" nodes are allowed to deliver an answer to lookup requests for a key. Several unexpected problems are discovered by formally analyzing HaeberlenPastry as discussed in Section 3.2.4, which leads to further improvements of the design of the Pastry protocol to IdealPastry and LuPastry. IdealPastry assumes that no nodes leave the network and that nodes do not join concurrently between two neighboring "ready" nodes. LuPastry relaxes these assumptions by only assuming no departure of nodes, under which condition it is proved to satisfy the correctness property *CorrectDelivery* in the presence of arbitrary concurrent joining of nodes. IdealPastry and LuPastry both realize the "dependable routing" correctness property of Pastry mentioned in Castro et al. (2004).

What does "dependable routing" formally mean? Does Pastry guarantee dependable routing? If so, to what extent, and how? Is there any fundamental design flaw in Pastry, in particular Castro et al. (2004), with respect to its correctness properties, such as dependable routing? If yes, how does the problem occur? If not, is there a formal proof?

This thesis formally describes dependable routing as correctness property *CorrectDelivery* of the system in Property 4.3.4. Unlike the original description stated in Castro et al. (2004) and repeated above, the formal definition of *CorrectDelivery* specifies that only "ready" nodes are qualified to answer a lookup messages for a key, which allows joining nodes to be closer to the key than the "ready" node delivering the key. "Closer" is formally defined as an ordering of the absolute distance between two nodes around the ring. Modular arithmetic is used to define different notions of distances in Definition 4.1.1. In order to solve the problem of two nodes with the same absolute distance to the key, Definition 4.1.2 defines asymmetric responsibility for a node w.r.t. some key according to clockwise (right) or counterclockwise (left) distance from node to key.

The guarantee of dependable routing relies heavily on how message exchange and updates of local states are implemented. As discovered in model checking analysis and discussed in Section 3.2, CastroPastry does not provide consistent local views of "ready" nodes in the presence of concurrent joins of new nodes between two "ready" nodes leading to violation of *CorrectDelivery*. HaeberlenPastry suffers also from the same difficulties with concurrent joining of many nodes within a closed region mentioned before. IdealPastry assumes that concurrent joining of new nodes between two "ready" nodes does not occur and *CorrectDelivery* is shown to be guaranteed in this case. Finally, LuPastry is verified to conform to *CorrectDelivery* all the time and in the presence of all possible message interleavings and concurrent joins to arbitrary configurations of "ready" nodes. However, none of the Pastry models introduced in this thesis can guarantee *CorrectDelivery* when departure of nodes is allowed in general.

Are there other interesting properties of such a system implementing a DHT? Are they interrelated?

Besides the safety property *CorrectDelivery*, other properties are analyzed in this thesis. For example, the fundamental invariant for proving *CorrectDelivery* is *NeighborClosest* (Definition 4.3.7), which guarantees a "ready" node will never ignore the existence of another "ready" node between itself and its direct neighbors. Several other invariants (Definition 6.5.2) are discovered to belong to the inductive invariant of LuPastry. In the proof sketch illustrated in Section 6.5.2, these invariants are used as proof arguments in the induction proof of *IRN*, which subsumes the invariant *NeighborClosest* by stating that no "ready" node r can be closer to an arbitrary node i than i's immediate neighbor.

Some interesting hypothetical properties are discovered to be false, such as *Symmetry* introduced in Property 5.2.3, which states that if a "ready" node has another "ready" node in its leaf set, then the other node also has this node in its leaf set. This is violated when a joining node is added by one "ready" node, but the notification message has not

yet arrived at the neighboring "ready" node. During the process of searching for invariants shown in Section 6.2.2, candidate invariants *IncludeNeighbor* and *CrossNeighbor* were analyzed. They basically state that between any two neighboring nodes, no other neighboring pair of nodes should exist, one of which is respectively inside or outside the region between the original pair. Violation of these hypothetical properties is illustrated in Figure 6.4, which shows how a node may have a right neighbor far away from it ignoring the closer "ready" nodes because it considers the node from its left neighborhood also as its right neighbor by the definition of *RightNeighbor*(*ls*) in Definition 4.1.2. However, these candidate invariants are in fact too strong and therefore are abandoned for the inductive proof of *NeighborClosest*.

Furthermore, it is also important to show that a system will eventually "do the right thing". Non-properties such as *NeverDeliver*, *NeverJoin*, *ConcurrentJoin* and *CcJoinDeliver* are proposed in this thesis to be proven by contradiction in order to show that Pastry is able to provide successful lookup, join, concurrent join, and lookup with concurrent join.

8.2 Answers to Questions About Methodology

How to formally model Pastry? What is a proper abstract level of modeling?

In this thesis, Pastry is modeled as a transition system in TLA$^+$, which provides the primitives both for the data structures of the static model and rule/message-based actions of the dynamic model.

Data structures are modeled using the primitives provided by TLA$^+$: e.g. node identifiers as intervals of the natural numbers; leaf sets of nodes as functional mappings from such intervals to records, where left and right leaf sets are modeled as sets of node identifiers; the complex routing table is modeled as a mapping from positions to values in a parameterized matrix; and messages are modeled as records.

The dynamic behaviors are modeled as actions, where timing-dependent actions are modeled to occur non-deterministically in order to simplify the formal model for tractability of model checking. However, message sending and receiving together with the necessary message content are modeled in detail in order to analyze the effect of all possible message interleavings in a network, which turns out to be crucial for the *CorrectDelivery* property of Pastry which can easily be violated as a consequence of message interleaving.

A formal specification of LuPastry is given in Chapter 4 and those of the other formal models of Pastry (CastroPastry, HaeberlenPastry and IdealPastry) are available online at VeriDis (2013).

How to express the correctness properties of Pastry in TLA$^+$? Is it at all feasible to prove their invariance considering the intricacies of message interleaving and complex data structures of Pastry, and how?

This thesis formally defines the correctness properties in Section 4.3 as theorems, including the type correctness of the model and its safety property *CorrectDelivery*. The invariance of the properties is expressed using a temporal operator provided by TLA$^+$, even though it is possible to prove it by induction.

The property *CorrectDelivery* is first reduced to *NeighborClosest* and *HalfNeighbor* as shown in Section 6.1. Neither of them is an inductive invariant of HAEBERLENPASTRY, which was first chosen as our initial verification target.

HalfNeighbor is then extended to become an inductive invariant in Section 6.2.1 where it is also proved. *NeighborClosest* is then shown to be violated due to *churning*. The protocol design of HAEBERLENPASTRY is therefore improved as shown in Section 3.2.4 to IDEALPASTRY.

The subtlest part of the theorem proving approach is to find and prove the proper inductive invariant of the algorithm. Assumptions are made, as introduced before, that no nodes leave the network and no nodes concurrently join the network between two neighboring “ready” nodes. Under these assumptions, a formal proof of IDEALPASTRY is constructed with help of TLC and TLAPS in an interactive way. The methodology of finding the proof is described in Section 1.2 on an abstract level and its challenges are stated in Section 1.3.6. Then more theoretical details are introduced in Section 2.2.2 with a flow chart in Figure 2.4. The methodology is finally explained in detail in Section 6.3 with a concrete example.

In the next step, assumptions of IDEALPASTRY are relaxed and the design is further improved to LUPASTRY. The proof for IDEALPASTRY is partially reused for proving invariance of *NeighborClosest* in LUPASTRY. However, the set of invariants and their proofs require modification as explained in Section 6.4.4. Final invariants and an example proof sketch are illustrated in Section 6.5.1, while a complete formal proof is available online in VeriDis (2013).

To what extent can the formal verification of such a system be conducted automatically?

Although the search for invariants and several important proof constructions must be designed by hand (Section 6.2), TLC can help to validate proposed invariants by automatically discovering counterexamples. TLAPS is used to interactively verify the proof. In TLAPS, a human conducts the proof sketch and describes the proof obligation and proof arguments, and postulates existence of a proof obtained by applying primitive proof rules with back-end provers. In case the proof cannot be found automatically for the postulated proof step, the proof obligation or arguments are modified, or the proof is broken down into sub-proof steps according to a different proof structure.

The proportion of automation of the proof of inductive invariants depends heavily on the power and reliability of the back-end prover. On one hand, a more powerful

back-end solver can reduce the proof time and the size of the formal proof sketch. For example, SMT solver is launched in more than 200 proof steps of the proof of invariants. Each of the proof step is a simplification of multiple proof steps using other theorem provers such as ZENON. On the other hand, only a proof proceeding in reliable small steps can be trusted because the minimal rules system is more tractable for humans. For example, TLAPS provides a feature for launching the core of ISABELLE to check the proof provided by ISABELLE/TLA. However, it is not yet possible to check the proof provided by an SMT solver in TLAPS.

When the complete proof sketch is iteratively verified, the whole proof can be checked with a single click in TLAPS, which altogether takes 4 hours using a Dell laptop with an Intel Core i3-2330M CPU at 2.2GHz with full usage of 8GB of memory.

8.3 Future Work

This thesis only considers safety properties of Pastry which guarantee the correctness property. The availability of successful routing and joining can be formulated as a liveness property, which can also be verified by discussing fairness conditions. Such an approach will be possible after TLAPS is developed to support the proof of liveness properties.

Node departure in Pastry may lead to network separation. Different proposals for handling departure of nodes can be found in the literature, in particular the active departure discussed in Section 7.2.2, by Lynch and Stoica (2004), Li et al. (2004) and Risson et al. (2005). The protocol for active departure of nodes seems to be similar to the concurrent join protocol, and it is not trivial to see if there can be any improvement of the protocol which guarantees the safety property under concurrent departure of arbitrary nodes (including those which are currently joining) together with concurrent joins.

The automated theorem provers at the back-end of TLAPS can be further improved to handle many trivial cases in the formal proof, which are currently broken into sub-proofs manually just because the formula is too complicated for the tools to provide a proof in time. Several such proof obligations in the inductive proof for LuPastry are automatically proved using SMT solvers, but there is no proof checker yet available for checking the result. In order to improve the automation, the proof can be inspected to find out which parts of the proof have the same pattern, that should be automatically proven using encoded simplification rules.

A more general framework for verification of a *DHT* can be constructed by generalization of the current TLA$^+$ model and its verification. As introduced in Section 7.3, the Ph.D. thesis Ghodsi (2006) can be consulted and compared for further details about such approach. Other approaches on Chord discussed in Section 7.2 can be formally modeled using TLA$^+$ and analyzed in a similar manner. The formal models CastroPastry, HaeberlenPastry and LuPastry can serve as templates for verification of similar distributed systems implementing a *DHT* and the general framework can be applied for other verification tasks.

Bibliography

Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, December 2004. ISSN 0360-0300. doi: 10.1145/1041680.1041681. URL `http://doi.acm.org/10.1145/1041680.1041681`.

Villu Arak. What happened on august 16. Blog, 2007. URL `http://heartbeat.skype.com/2007/08/what_happened_on_august_16.html`.

Leo Bachmair, Harald Ganzinger, Christopher Lynch, and Wayne Snyder. *Basic paramodulation and superposition.* Springer, 1992.

Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

Rana Bakhshi and Dilian Gurov. Verification of peer-to-peer algorithms: A case study. *Electr. Notes Theor. Comput. Sci.*, 181:35–47, 2007.

Clark Barrett and Cesare Tinelli. Cvc3. In *Computer Aided Verification*, pages 298–302. Springer, 2007.

Mordechai Ben-Ari. *Mathematical logic for computer science.* Springer, 2012.

Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Springer-Verlag New York Incorporated, 2004.

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs.* Springer, 1999.

Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *9th Workshop on Hot Topics in Operating Systems (HotOS), Lihue, Hawaii*, volume 18, page 21, 2003.

Niels Boeing. Das neue Internet. Zeit Online, 2012. URL `http://www.zeit.de/zeit-wissen/2012/05/Das-alternative-Netz`.

Francesco A. Bongiovanni. *Design, Formalization and Implementation of Overlay Networks; Application to RDF Data Storage.* PhD thesis, Ecole doctorale STIC science et technologies de l'information et de la communication, 2012.

Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 151–165. Springer, 2007.

Johannes Borgström, Uwe Nestmann, Luc Onana Alima, and Dilian Gurov. Verifying a structured peer-to-peer overlay network: The static case. In Corrado Priami and Paola Quaglia, editors, *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2004. ISBN 3-540-24101-9.

Matthew Caesar, Miguel Castro, Edmund B. Nightingale, Greg O'Shea, and Antony Rowstron. Virtual ring routing: network routing inspired by DHTs. *SIGCOMM Comput. Commun. Rev.*, 36(4):351–362, 2006. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/1151659.1159954.

Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. In *Microsoft Technical report MSR-TR-2003-52*, 2003.

Miguel Castro, Manuel Costa, and Antony I. T. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *International Conference on Dependable Systems and Networks (DSN 2004)*, pages 9–18, Florence, Italy, 2004. IEEE Computer Society.

Robert N. Charette. Why software fails, 2005. URL `http://spectrum.ieee.org/computing/software/why-software-fails`.

Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA^+ proof system. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148. Springer, 2010. ISBN 978-3-642-14202-4.

Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418. ACM, 2003.

Edmund M Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26. Springer, 2008.

Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

Edmund M Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *Computer Aided Verification*, pages 450–462. Springer, 1993.

Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

European Commission. P2P comes to the rescue of internet video, 06 2012. URL http://ec.europa.eu/information_society/newsroom/cf/itemdetail.cfm?item_id=8219.

Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA^+ proofs. In *FM 2012: Formal Methods*, pages 147–154. Springer, 2012.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

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 *ACM Symposium on Operating Systems Principles: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, volume 14, pages 205–220, 2007.

Nachum Dershowitz. Software horror stories, 2013. URL http://www.cs.tau.ac.il/~nachumd/horror.html.

Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

E Allen Emerson and Joseph Y Halpern. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1): 151–178, 1986.

Ghofrane Fersi, Wassef Louati, and Maher Ben Jemaa. Distributed hash table-based routing and data management in wireless sensor networks: a survey. *Wireless Networks*, 19(2):219–236, 2013.

Melvin Fitting. *First-order logic and automated theorem proving*. Springer Verlag, 1996.

FreePastry. Pastry: A substrate for peer-to-peer applications, 2009. URL http://www.freepastry.org/. Rice University and Max-Planck Institute for Software Systems.

Ali Ghodsi. *Distributed k-ary system: Algorithms for distributed hash tables*. PhD thesis, KTH-Royal Institute of Technology, 2006.

Mark Gillett. What does skype's architecture do?, 2012. URL http://blogs.skype.com/2012/07/26/what-does-skypes-architecture-do/#fbid=-7_uyCfdt2m.

Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification*, pages 176–185. Springer, 1991.

Dan Goodin. Skype replaces P2P supernodes with linux boxes hosted by microsoft (updated), 2012. URL `http://arstechnica.com/business/2012/05/skype-replaces-p2p-supernodes-with-linux-boxes-hosted-by-microsoft/`.

Orna Grumberg and Helmut Veith. *25 years of model checking: history, achievements, perspectives*, volume 5000. Springer-Verlag New York Incorporated, 2008.

K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 381–394, New York, NY, USA, 2003. ACM. ISBN 1-58113-735-4. doi: 10.1145/863955.863998. URL `http://dx.doi.org/10.1145/863955.863998`.

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.

Elizabeth Heichler and Jared Newman. Skype blames outage on "supernode" problem, 2010. URL `http://www.itbusiness.ca/news/skype-blames-outage-on-supernode-problem/14322`.

Joseph M Hellerstein. Toward network data independence. *ACM SIGMOD Record*, 32 (3):34–40, 2003.

Michael Gerard Hinchey and Jonathan Peter Bowen. *Industrial-strength formal methods in practice*. Springer Verlag, 1999.

IEEE. IEEE guide–adoption of the project management institute (pmi(r)) standard a guide to the project management body of knowledge (pmbok(r) guide)–fourth edition. *IEEE Std 1490-2011*, pages 1–508, 2011. doi: 10.1109/IEEESTD.2011.6086685.

Martin Karsten, S. Keshav, Sanjiva Prasad, and Mirza Beg. An axiomatic basis for communication. In Jun Murai and Kenjiro Cho, editors, *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Kyoto, Japan, August 27-31, 2007*, pages 217–228. ACM, 2007. ISBN 978-1-59593-713-1. doi: http://doi.acm.org/10.1145/1282380.1282405.

L. Lamport. TLA tools, 2012a. URL `http://www.tlaplus.net/`.

Leslie Lamport. Sometime is sometimes not never: On the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 174–185. ACM, 1980.

Leslie Lamport. A summary of TLA$^+$, 06 2000. URL `http://research.microsoft.com/en-us/um/people/lamport/tla/summary.pdf`.

Leslie Lamport. *Specifying Systems, The TLA$^+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X.

Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, 2012b.

Daniel Lehmann and Michael O. Rabin. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '81, pages 133–138, New York, NY, USA, 1981. ACM. ISBN 0-89791-029-X. doi: 10.1145/567532.567547. URL `http://doi.acm.org/10.1145/567532.567547`.

Xiaozhou Li, Jayadev Misra, and C Greg Plaxton. Active and concurrent topology maintenance. In *Distributed Computing*, pages 320–334. Springer, 2004.

Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 395–406. ACM, 2003.

Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Model checking the Pastry routing protocol. In Jens Bendisposto, Michael Leuschel, and Markus Roggenbach, editors, *10th Intl. Workshop Automatic Verification of Critical Systems (AVOCS)*, pages 19–21, Düsseldorf, Germany, September 2010. Universität Düseldorf.

Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Towards verification of the pastry protocol using TLA$^+$. Research Report MPI-I-2011-RG1-002, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, June 2011.

Nancy Lynch and Ion Stoica. Multichord: A resilient namespace management protocol. 2004.

marketerterryb. Skype outage today, 12 2010. URL `http://blogs.skype.com/2010/12/22/skype-outage-today/#fbid=gTdeoNSvnZ1`.

Kenneth L McMillan. *Symbolic model checking*. Springer, 1993.

Stephan Merz. The specification language TLA$^+$. In Dines Bjørner and Martin C. Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science, pages 401–451. Springer, Berlin-Heidelberg, 2008.

Microsoft. About skype: What are p2p communications?, 2013. URL `https://support.skype.com/en/faq/fa10983/what-are-p2p-communications`.

Dejan S Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-peer computing, 2002.

Peter G. Neumann. Illustrative risks to the public in the use of computer systems and related technology. Webpage, 04 2013. URL `http://www.csl.sri.com/users/neumann/illustrativerisks.html`.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

Andy Oram. *Peer-to-peer: harnessing the benefits of a disruptive technology.* O'Reilly Media, Inc., 2001.

Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5, 1989.

Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *Computer aided verification*, pages 377–390. Springer, 1994.

Virgile Prevosto and Uwe Waldmann, 2006. URL `http://www.mpi-inf.mpg.de/~uwe/paper/TSPASS-bibl.html`.

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*, volume 31. ACM, 2001.

Sean Christopher Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. *Handling churn in a DHT.* Computer Science Division, University of California, 2003.

Alexandre Riazanov and Andrei Voronkov. Vampire 1.1 (system description). In *Proceedings of the First International Joint Conference on Automated Reasoning*, IJCAR '01, pages 376–380, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42254-4. URL `http://dl.acm.org/citation.cfm?id=648237.753936`.

John Risson and Tim Moors. Survey of research towards robust peer-to-peer networks: Search methods. *Computer Networks*, 50(17):3485 – 3521, 2006. ISSN 1389-1286. doi: 10.1016/j.comnet.2006.02.001. URL `http://www.sciencedirect.com/science/article/pii/S1389128606000223`.

John Risson, Ken Robinson, and Tim Moors. Fault tolerant active rings for structured peer-to-peer overlays. In *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, pages 18–25. IEEE, 2005.

Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10): 72–82, 2010. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1831407.1831427.

Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference*

on Distributed Systems Platforms (Middleware), pages 329–350, November 2001. URL `http://www.freepastry.org/PAST/pastry.pdf`.

Rüdiger Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101–102. IEEE, 2001.

Stephan Schulz. E-a brainiac theorem prover. *AI Communications*, 15(2):111–126, 2002.

Carl-Johann Seger. *An introduction to formal hardware verification.* University of British Columbia, Department of Computer Science, 1992.

Ralf [Hrsg.] Steinmetz, editor. *Peer-to-peer systems and applications.* Lecture notes in computer science ; 3485. Springer, Berlin, 2005. ISBN 3-540-29192-X; 978-3-540-29192-3. URL `http://link.springer.com/book/10.1007/11530657`.

Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 149–160. ACM, 2001.

Antti Valmari. A stubborn attack on state explosion. In *Computer-Aided Verification*, pages 156–165. Springer, 1991.

VeriDis. The TLA$^+$ codes for the pastry model, 2013. URL `http://www.mpi-inf.mpg.de/~tianlu/fmPastry/`.

Christoph Weidenbach, Bernd Gaede, and Georg Rock. Spass & flotter version 0.42. In *CADE*, pages 141–145, 1996.

Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In *Automated Deduction–CADE-22*, pages 140–145. Springer, 2009.

Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. The isabelle framework. In *Theorem Proving in Higher Order Logics*, pages 33–38. Springer, 2008.

Dan York. Understanding today's skype outage: Explaining supernodes, 2010. URL `http://www.disruptivetelephony.com/2010/12/understanding-todays-skype-outage-explaining-supernodes.html`.

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer.

Pamela Zave. Using lightweight modeling to understand chord. *Computer Communication Review*, 42(2):49–57, 2012.

Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.