A Machine-Checked Correctness Proof for Pastry

Noran Azmy[a,b], Stephan Merz[b,*], Christoph Weidenbach[a]

[a] *Max Planck Institute for Informatics*
[b] *University of Lorraine, CNRS, Inria, LORIA, Nancy, France*

Abstract

Protocols implemented on overlay networks in a peer-to-peer (P2P) setting promise flexibility, performance, and scalability due to the possibility for nodes to join and leave the network while the protocol is running. These protocols must ensure that all nodes maintain a consistent view of the network, in the absence of centralized control, so that requests can be routed to the intended destination. This aspect represents an interesting target for formal verification. In previous work, Lu studied the Pastry algorithm for implementing a distributed hash table (DHT) over a P2P network and identified problems in published versions of the algorithm. He suggested a variant of the algorithm, together with a machine-checked proof in the TLA$^+$ Proof System (TLAPS), assuming the absence of node failures. We identify and correct problems in Lu's proof that are due to unchecked assumptions concerning modulus arithmetic and underlying data structures. We introduce higher-level abstractions into the specifications and proofs that are intended for improving the degree of automation achieved by the proof backends. These abstractions are instrumental for presenting the first complete formal proof. Finally, we formally prove that an even simpler version of Lu's algorithm, in which the final phase of the join protocol is omitted, is still correct, again assuming that nodes do not fail.

Keywords: formal verification, machine-checked proof, TLA+, distributed algorithm, peer-to-peer protocol, distributed hash table

1. Introduction

In a peer-to-peer (P2P) network, individual nodes, or *peers*, communicate directly with each other and act as both suppliers and users of a given service. P2P networks are motivated by their self-organization, scalability and robustness, since there is no central server representing a single point of failure or a performance bottleneck.

A key problem with P2P networks—particularly large-scale ones—is how to efficiently manage the available resources. In completely unstructured P2P networks where no topology is imposed on the nodes, functions like search typically resort to flooding the network via broadcasting a search request until the request reaches a peer that has the required data item. Flooding causes a very high amount of unnecessary network traffic, as well as CPU and memory usage [1].

*Corresponding author.

Distributed hash tables (DHTs) are a way of structuring P2P networks so that the available resources are organized, and communication among peers is reliable and efficient. A DHT implements a hash table where key-value pairs are stored at different nodes on the network. Nodes are assigned unique identifiers, and messages from one node to another—instead of being flooded through the network—are routed through a number of intermediate nodes, the route being determined by node identifiers. Distributed hash tables tap the advantages of both P2P communication and hash tables, with a simple and elegant design that enables locating a required piece of data with high efficiency, and without the need for global information. As in a classic hash table, the main function of a DHT is key lookup. Due to the lack of a central server with a global view of the network, nodes in a DHT must collaborate to decide on their respective key storage range, and to route lookup requests to the appropriate node. Pastry, Chord, Kademlia, CAN and $\mathcal{DKS}$ are among the most popular published DHT protocols [2, 3, 4, 5, 6]. These protocols are similar in that they focus on the efficient management of data stored in a distributed fashion over a large number of nodes, but they differ in some characteristics such as the topology of the overlay network, the distance function between nodes on the network, and the routing model.

In most practical applications, P2P/DHT networks are subject to a certain level of *churn*; nodes are continuously joining and leaving the network, and they may fail abruptly without giving notice to other nodes. The DHT implementation should handle this turbulence efficiently and smoothly and ensure that the network always recovers to a stable state where connectivity among the live nodes is maintained and there is no confusion among the nodes about the key space. This aspect presents an interesting target for formal verification.

In this paper, we present the results of our formal verification of Pastry. We verify correct delivery of lookups for two variants of Pastry by giving two complete proofs of correctness written in the interactive proof assistant TLAPS [7]. It has already been shown in [8] that published variants of the protocol violate this correctness property: not only may node departures and failures may cause the network to separate irreversibly, but more surprisingly, the Pastry ring may even disorganize when new nodes join the network. Here, we show that a version of Pastry suggested by Lu [9] is correct with respect to delivery of lookup messages in a pure-join model, i.e., where no nodes may leave the network or fail. We also show that in the pure-join model, correctness still holds using a join protocol that is simpler than that used by Lu and originally proposed in [10].

1.1. Related Work

Bakhshi et al. [11] describe an abstract model for structured P2P networks with a ring topology in the π-calculus, and use this model for verifying the stabilization algorithm of Chord by establishing weak bisimulation between the specification of Chord as a ring network and the implementation of the stabilization algorithm. This is a pure-join model in which node failure is not taken into account, and features such as finger (routing) tables and node successor lists are not modeled. Using Alloy to formally model and verify Chord, Zave [12] shows that the pure-join Chord protocol is correct, but that the full version of the protocol may not maintain the claimed invariants. In subsequent work [13] she presents a full version of Chord (where both node arrivals and departures are

modeled) with a partly mechanized proof of correctness. The correctness of this version of Chord relies on the assumption that there is a *stable base* of $r+1$ permanent network members, where r is the size of the successor list maintained by each node. The authors of the protocol $\mathcal{DKS}$ conduct some experiments using simulation to observe how lookup efficiency is affected by churn [6]. In his Ph.D. thesis, Ghodsi [14] discusses several issues such as concurrent joins and node failure, and claims that it is impossible to guarantee correctness when node failure is possible, due to the possibility of network separation. Borgström et al. [15] use CCS for the formal verification of lookups in the static case of the protocol, i.e., without taking node joins or failure into account.

The work that is most relevant to this paper was done by Lu on Pastry [8, 9, 16]. Lu models Pastry in TLA$^+$, and uses the TLC model checker and the TLAPS proof assistant to formally verify *correct delivery* of lookups: *at any point in time, there is at most one node that answers a lookup request for a key, and this node must be the closest live node to that key.* As in the case of Chord, Lu discovers several problems in the original Pastry protocol. He also shows that the improvements proposed in later publications on Pastry, in particular by Haeberlen et al. [10], still do not guarantee correct delivery, even in the absence of node failures. Finally, he presents a pure-join variant of Pastry, which he calls LuPastry, for which he verifies correct delivery. Notably, Lu's Pastry variant restricts the protocol described in [10] by enforcing that a live node may only facilitate the joining of one newly arriving node at a time. Lu's proof reduces correct delivery to a set of around 50 claimed invariants, which are proved with the help of TLAPS. As such, LuPastry represents a major effort in the area of computer-aided formal verification of distributed algorithms. Due to the sheer size of the proof, however, as well as the lack of maturity of the tools at the time, Lu's proof relies on many unproved assumptions relating to arithmetic and to protocol-specific data structures. Upon examining Lu's proof, we discovered counterexamples to several of the underlying assumptions. While we were able to prove weaker variants of many assumptions, this was not possible for others. In fact, we were able to find a counterexample to one of Lu's claimed invariants, for which the TLA$^+$ proof was only possible because of incorrect assumptions. This led us to redesign the overall proof of correctness for Pastry. In the process, we introduced higher-level abstractions in the specification and the proof that help make the TLA$^+$ specification of the protocol more understandable and, importantly, also help improve the degree of automation of the proof.

Our improved specifications and the outline of the new proof were published in [17], and the present article is an extended version of that conference paper that contains a more detailed presentation of our contributions. Moreover, we observe that the node join process of the protocol can be simplified substantially, without impacting correctness: the invariants used for the proof reveal that the final "lease exchange" step, a handshaking step between a new node and its neighbor nodes before it becomes an active participant, is not necessary for correctness in the join-only scenario. In fact, this step was not part of the original Pastry protocol published in [2], but was introduced by Haeberlen et al. [10], among other improvements to the protocol. Although the reasons for adding the lease exchange step are not stated explicitly, one may suspect that it was introduced in order to improve the accuracy of the leaf sets and, consequently, the consistency of lookups in the protocol. Lu shows, however, that this lease exchange step does not guarantee lookup consistency: the full

dynamic protocol where nodes join and leave freely violates correct delivery of lookup messages, even with the implementation of lease exchange. We observe, on the other hand, that the pure-join model is correct without this step. We formalize a variant of the LuPastry specification where the lease exchange step is omitted, and we prove correct delivery of messages for this simpler variant.

1.2. Contribution

The contribution of this paper is threefold. First, we analyze the data-level assumptions underlying Lu's proof and replace the relevant ones by theorems that we prove using TLAPS. Second, we present an improved TLA$^+$ specification of LuPastry, which we denote by LuPastry$^+$, and provide a complete proof of correct delivery for LuPastry$^+$ in TLA$^+$. The improved specification supports further proof automation by the introduction of an intermediate specification layer and systematic lemma support for TLA$^+$ CHOOSE expressions. Third, we show that the lease exchange phase of the node join process is not needed for achieving correctness in the pure-join model. We present a simplified node join process for LuPastry$^+$ where lease exchange is omitted, and reuse our TLA$^+$ proof for the original version of the protocol to prove correct delivery of lookup messages for the simplified protocol. We refer to the new protocol as Simplified LuPastry$^+$. The formal specification of the simplified protocol is obtained by the obvious modifications from LuPastry$^+$, and the proof only requires similar minor changes, resulting in a pleasant experience of robustness. Although we formally only establish that the simpler protocol is correct for the pure-join model, Lu's previous results indicate that the lease exchange phase is also of little help in a setting that tolerates node failure.

Outline of the paper. Section 2 introduces the Pastry algorithm, and in particular its protocol for integrating new nodes, as well as the TLA$^+$ specification formalism. Our formal model of LuPastry$^+$ and an outline of its correctness proof appear in Sections 3 and 4. Section 5 presents the proposed simplification of the join protocol and the adaptation of the correctness proof. Finally, Section 6 concludes the paper and indicates future work.

2. Background

2.1. The Pastry Algorithm

We first give a short, informal introduction to the Pastry algorithm, focusing on the protocol for integrating new nodes. More detailed information on Pastry can be found in the original publication introducing the algorithm [2].

The Pastry network can be visualized as a ring of keys with identifiers in the set $I \triangleq 0 \,..\, (2^M - 1)$ for some positive integer M (see Figure 1). Nodes participating in the protocol are assigned unique node IDs drawn from the same set I, and live nodes (indicated by black circles in Figure 1) *cover* keys that are numerically close to their node IDs. The *coverage* of node i is a contiguous range of keys, including i, and for all keys k in that set, i considers itself (a) the proper recipient of all look-up messages addressed to k, and (b) the node responsible for facilitating the joining of any new node with ID k. In the absence of a central server and shared memory, live nodes need to rely on message passing and local information to agree on a proper division of coverage.

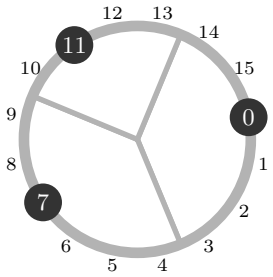


Figure 1: A Pastry ring of size 16 with three live nodes 0, 7 and 11.

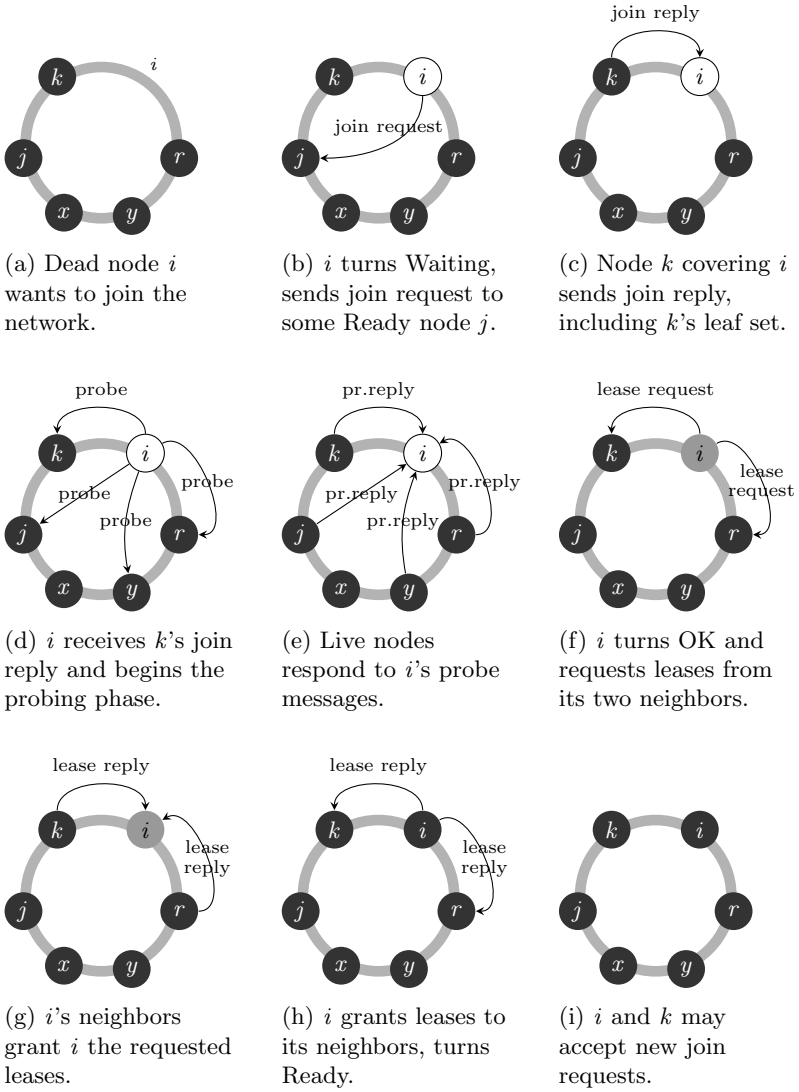
Let *Ready* denote the set of live nodes that have been fully integrated in the Pastry ring. Ready nodes are of particular interest since only Ready nodes may accept look-up messages or help new nodes join the ring. Ideally, the coverage ranges computed by all Ready nodes (a) do not overlap, (b) cover the whole range of keys, and (c) are computed based on the smallest absolute distance to the node: if a Ready node i covers key k, then k is closer to i in terms of absolute ring distance than it is to any other Ready node $j \neq i$, with a rule for breaking ties. These conditions all hold for the ring illustrated in Figure 1. Condition (b) may be temporarily violated when a new node joins but is not yet Ready; it is therefore required to hold only if there are no nodes in the process of joining (i.e., if all live nodes in the ring are Ready).

A node i computes its coverage by maintaining a *leaf set*: a set containing what i believes to be its L live neighbor nodes on both sides, where the positive integer L is a parameter of the algorithm, denoting the size of the leaf set on either side. Larger values of L may be expected to offer better network connectivity and robustness against node failure, at the cost of more maintenance overhead.[1] The left and right neighbors of node i are the two members of its leaf set that are closest to node i on either side, and its coverage is the interval between the midpoints between its own ID i and the IDs of its neighbors. In Figure 1, assuming up-to-date leaf sets, the left and right neighbors of node 0 are nodes 11 and 7, respectively. Therefore, its coverage is the interval $[14, 3]$ (*i.e.*, the set of keys $\{14, 15, 0, 1, 2, 3\}$).

The join protocol of Pastry is responsible for maintaining up to date the leaf sets (and thus the coverage) of nodes whenever new nodes join the network. We present here the join protocol as suggested by Lu, which imposes that any live node may help at most one node join the ring at any time. Also following Lu, our proof assumes that nodes do not fail or leave the ring, and therefore the set of live nodes can only grow during the execution of the protocol.

The protocol is illustrated in Figure 2. Nodes are either *Dead* (not shown in the figure), *Waiting* (white), *OK* (gray) or *Ready* (black). Only Ready nodes facilitate new nodes joining the network. A Dead node i that decides to join the network turns to Waiting and sends a *join request* to a Ready node j that it knows about. The request is forwarded to the Ready node k that covers key i.

[1] Nodes also maintain routing tables for the purpose of efficient message routing, but these are not important for our discussion.



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

(b) i turns Waiting, sends join request to some Ready node j.

(c) Node k covering i sends join reply, including k's leaf set.

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

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

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

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

(h) i grants leases to its neighbors, turns Ready.

(i) i and k may accept new join requests.

Figure 2: The Pastry Join Protocol.

Node k responds to i's request when it is free for handling a new join request. As part of its response, node k communicates its own leaf set to node i as a seed for constructing i's leaf set. Node i receives k's reply and, in order to construct its proper leaf set, sends *probe* messages to the nodes in the leaf set that it received from k. (In this figure, we assume that $L = 2$.) All live nodes that receive the probe add i to their leaf set if appropriate (i.e., if node i is among the L closest live neighbor nodes), and send a *probe reply* to i with their own leaf set information. This process continues until node i has received probe replies from all nodes it has heard about and that are close enough to i to be in i's leaf set, after which i becomes OK. In order for node i to become Ready and eventually serve the IDs closest to it, node i has to exchange *leases* with both its left and right neighbors (one of which must be k). Node i sends out *lease*

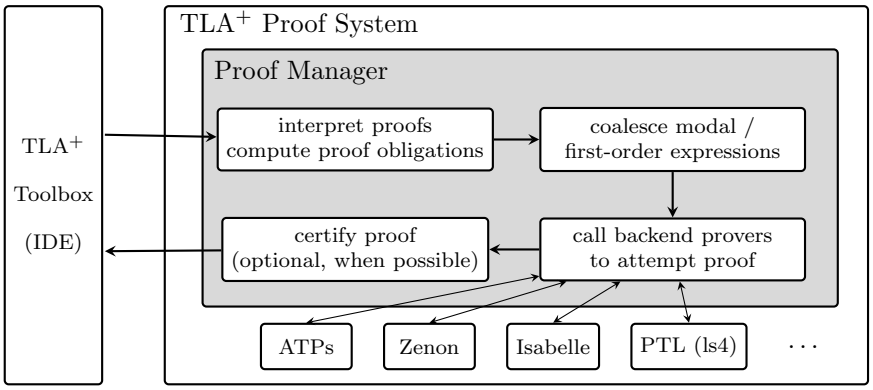


Figure 3: The TLA$^+$ Proof System

request messages to both of its neighbors. If i's neighbor is Ready or OK, and also considers i to be its neighbor, it grants i the lease in a *lease reply* message. When node i has received lease replies from both its neighbors, it switches to Ready, and in turn grants leases to its neighbors. When node k receives i's lease, it knows that the node has joined successfully and may subsequently help other nodes join the ring.

2.2. The TLA$^+$ Specification Language and Tools

TLA$^+$ [18] is a formal specification language that mainly targets concurrent and distributed algorithms and systems. It is based on untyped Zermelo-Fraenkel set theory for specifying data structures, and on the Temporal Logic of Actions, a variant of linear-time temporal logic, for describing system behavior. Systems are specified as state machines over a tuple of state variables by defining a state predicate *Init* and a transition predicate *Next* that constrain the possible initial states and the next-state relation. Transition predicates (also called *actions*) are first-order formulas that contain unprimed and primed state variables for denoting the values of the variables in the state before and after the transition. Progress can be ensured by specifying fairness hypotheses that require certain actions to be taken if they are enabled "often enough". Fairness hypotheses are irrelevant for proving safety properties and will not play a role in the proof presented here.

The main tools for verifying properties of TLA$^+$ specifications are TLC [19], an explicit-state model checker for finite instances of TLA$^+$ specifications, and TLAPS, the TLA$^+$ Proof System [7]. TLAPS is based on a hierarchical proof language; the user writes a TLA$^+$ proof in the form of a hierarchy of *proof steps*, each of which is interpreted by the *proof manager*, which generates corresponding proof obligations and passes them to back-end provers. These include automatic theorem provers (ATPs, both SMT solvers [20, 21, 22] and superposition-based provers such as SPASS [23]), Zenon [24], Isabelle/TLA$^+$, a faithful encoding of TLA$^+$ set theory in the proof assistant Isabelle [25], and a decision procedure for propositional temporal logic [26] (see Figure 3). Larger steps that cannot be proved directly by any of the back-end provers can be broken down into sub-steps. Because the language is untyped, part of the proof effort consists in proving a *typing invariant* that delimits the sets of values that

variables assume during any execution of the state machine, and in particular the domains and co-domains of functions and user-defined operators.

The user interacts with the TLA$^+$ specification and tools through the TLA$^+$ Toolbox, an IDE based on Eclipse. Since the model checker and the proof assistant operate on the same specification, it is possible and recommended to extensively validate a specification by using TLC for checking properties and non-properties over finite instances before starting to write interactive proofs.

3. Specifying Pastry in TLA$^+$

We give an overview of our specification of Pastry in TLA$^+$.[2] Our specification, which we call LuPastry$^+$, is derived from Lu's specification [8]. The differences between the two specifications are outlined in Section 3.3.

3.1. Static Model

The specification is parameterized by the positive integer M. We define $\mathit{RingSize} \triangleq 2^M$; the interval $I \triangleq 0 \,..\, (\mathit{RingSize} - 1)$ is the set of key and node IDs. The parameters L and A denote respectively the size of the leaf sets, i.e., the number of neighbors on either side stored by each node, and the set of nodes that are assumed to be live initially (all these nodes will be Ready, with up-to-date leaf sets).

For two nodes $x, y \in I$ we define the *clockwise* distance as well as the *absolute* distance, *i.e.* the shortest distance between these nodes.

$$
\begin{array}{l}
\mathit{ClockwiseDistance}(x, y) \triangleq \\
\quad \text{IF } x \leq y \text{ THEN } y - x \text{ ELSE } \mathit{RingSize} - x + y
\end{array}
$$

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

In order to minimize the number of arithmetic comparisons and calculations in the subsequent definitions, we define a ternary predicate $\mathit{ClockwiseArc}(x, y, z)$ that holds if node y lies on the clockwise arc connecting x and z along the ring.

$$
\begin{array}{l}
\mathit{ClockwiseArc}(x, y, z) \triangleq \\
\quad \mathit{ClockwiseDistance}(x, y) \leq \mathit{ClockwiseDistance}(x, z)
\end{array}
$$

Given a node x and a set S of nodes, the following operator designates the closest node in S to the right of x (or x itself if $S = \emptyset$). The CHOOSE operator of TLA$^+$ denotes Hilbert's choice. More precisely, $\text{CHOOSE } x \in S : P(x)$ denotes some element $x \in S$ satisfying the predicate P if such an element exists, and a fixed arbitrary value otherwise.

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

[2] The specification is available at `https://members.loria.fr/SMerz/projects/pastry/`.

The closest node to the left is defined symmetrically. More generally, the operator $ClosestNodesFromTheRight(x, S, n)$, defined in a similar way, identifies the set of at most n elements drawn from S that are the closest right neighbors of x, and similarly for the leftmost neighbors.

The leaf set data structure is represented as a record that contains the owner of the leaf set and two sets of at most L nodes containing what the owner believes to be the closest live nodes in the left and right neighborhoods.

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

The following operators access the contents of a leaf set and retrieve the right neighbor of a node, based on the information contained in its leaf set. Again, the left neighbor is defined symmetrically.

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

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

A leaf set ls is *proper* if all nodes in $ls.left$ (resp., $ls.right$) are to the left (resp., right) of $ls.node$. Some care has to be taken in the formal definition in order to encompass border cases of a ring with few active nodes, where the "left" and "right" parts of the leaf set may overlap.

$$\begin{aligned} &IsProper(ls) \triangleq \\ &\quad \land\ \forall x \in ls.left \setminus ls.right,\ y \in ls.right : ClockwiseArc(y, x, ls.node) \\ &\quad \land\ \forall x \in ls.right \setminus ls.left,\ y \in ls.left : ClockwiseArc(ls.node, x, y) \end{aligned}$$

The following definitions introduce the left and right bounds of the coverage interval of a node, based on its leaf set. Observe that there is a slight asymmetry between the two definitions, in order to break the tie at the midpoint between two nodes. A key is covered if it lies on the arc connecting the bounds of the interval.

$$\begin{aligned} &LeftCoverage(ls) \triangleq \\ &\quad \text{IF } LeftNeighbor(ls) = ls.node \text{ THEN } ls.node \\ &\quad \text{ELSE } (LeftNeighbor(ls) + \\ &\qquad (ClockwiseDistance(LeftNeighbor(ls), ls.node) \div 2 + 1)) \% RingSize \end{aligned}$$

$$\begin{aligned} &RightCoverage(ls) \triangleq \\ &\quad \text{IF } RightNeighbor(ls) = ls.node \\ &\quad \text{THEN } (RingSize + ls.node - 1) \% RingSize \\ &\quad \text{ELSE } (ls.node + \\ &\qquad ClockwiseDistance(ls.node, RightNeighbor(ls)) \div 2) \% RingSize \end{aligned}$$

$$Covers(ls, k) \triangleq ClockwiseArc(LeftCoverage(ls), k, RightCoverage(ls))$$

Leaf sets are constructed from the empty leaf set (containing only its owner) by adding nodes, as described by the following operators.

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

$$
\begin{array}{lll}
AddToLS(S, ls) \triangleq & \text{LET } x \triangleq ls.node & \\
& \quad cands \triangleq (ls.left \cup ls.right \cup S) \setminus x & \\
& \quad newleft \triangleq ClosestNodesFromTheLeft(x, cands, L) & \\
& \quad newright \triangleq ClosestNodesFromTheRight(x, cands, L) & \\
& \text{IN } [node \mapsto x,\ left \mapsto newleft,\ right \mapsto newright] &
\end{array}
$$

Similar definitions are introduced for the data structure of routing tables. Because these are irrelevant for our correctness proofs, we omit them here.

3.2. Specifying Executions of Pastry

Configurations of the Pastry network are represented using the following state variables.

$$vars \triangleq \langle Messages,\ Status,\ LeafSets,\ Probing,\ Leases,\ Grants,\ ToJoin \rangle$$

The variable *Messages* represents the set of messages that have been sent but not yet received. Messages are added to this set by the sending node and removed when they are handled by the destination node. Variables *Status* and *LeafSets* are arrays (i.e., functions) whose i-th entries are the current status, e.g., "Dead" or "Waiting", and leaf set of node i, respectively. Similarly, $Probing[i]$ is the set of nodes that node i has probed but has not heard back from yet, $Leases[i]$ and $Grants[i]$ are the set of nodes that node i has acquired leases from, and granted leases to, respectively. Lastly, $ToJoin[i]$ designates the node that is currently joining through i, if any, otherwise $ToJoin[i] = i$. Our full specification also models the routing tables used in Pastry for efficient message delivery. However, it is well known [2, 3] that these are not relevant for the correctness of the algorithm, and our proof confirms this because the only fact that we use is a type correctness predicate. We therefore omit routing tables from the presentation in this paper. For better readability, we have also renamed some of the operators that appear in our TLA$^+$ specification.

Executions of Pastry are specified through a state machine represented by the operators *Init* and *Next* (cf. Figure 4) that describe the initial state and the next-state relation. The overall TLA$^+$ specification is defined as the formula

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

The disjuncts of formula *Next* describe the individual transitions (also known as actions) executed by nodes in the Pastry protocol.

The actions $Lookup(i, j)$ and $RouteLookup(i, j)$ model the events in which node i sends or forwards a lookup message for key j, respectively. Action $DeliverLookup(i, j)$ describes a ready node i receiving a lookup message for some key j that it covers. *Join*, *RouteJoinRequest* and *ReceiveJoinRequest* define similar events for join request messages. Action $ReceiveJoinReply(i)$ models a waiting node i receiving a reply to its join request and starting the probing process, which is modeled by the actions *ReceiveProbe* and *ReceiveProbeReply*. Finally, the actions *RequestLease*, *ReceiveLeaseRequest* and *ReceiveLeaseReply* correspond to the lease exchange protocol that occurs after probing is finished. Observe that our specification models asynchronous communication through distinct send and receive actions for each kind of message, and that messages can be arbitrarily delayed. Since we are only interested in safety properties, the specification does not enforce that messages are eventually received. In

$$
\begin{array}{l}
\mathit{Init} \triangleq \\
\quad \land\ \mathit{Messages} = \{\} \\
\quad \land\ \mathit{Status} = [i \in I \mapsto \text{IF } i \in A \text{ THEN } \text{“Ready”} \text{ ELSE } \text{“Dead”}] \\
\quad \land\ \mathit{LeafSets} = [i \in I \mapsto \text{IF } i \in A \text{ THEN } \mathit{AddToLS}(A, \mathit{EmptyLS}(i)) \\
\qquad\qquad\qquad\qquad \text{ELSE } \mathit{EmptyLS}(i)] \\
\quad \land\ \mathit{Probing} = [i \in I \mapsto \{\}] \\
\quad \land\ \mathit{Leases} = [i \in I \mapsto \text{IF } i \in A \text{ THEN } A \text{ ELSE } \{i\}] \\
\quad \land\ \mathit{Grants} = [i \in I \mapsto \text{IF } i \in A \text{ THEN } A \text{ ELSE } \{i\}] \\
\quad \land\ \mathit{ToJoin} = [i \in I \mapsto i] \\
\\
\mathit{Next} \triangleq \exists i, j \in I : \\
\quad \lor \mathit{Lookup}(i,j) \qquad\qquad \lor \mathit{RouteLookup}(i,j) \\
\quad \lor \mathit{DeliverLookup}(i,j) \qquad \lor \mathit{Join}(i,j) \\
\quad \lor \mathit{RouteJoinRequest}(i,j) \qquad \lor \mathit{ReceiveJoinRequest}(i) \\
\quad \lor \mathit{ReceiveJoinReply}(i) \qquad \lor \mathit{ReceiveProbe}(i) \\
\quad \lor \mathit{ReceiveProbeReply}(i) \qquad \lor \mathit{RequestLease}(i) \\
\quad \lor \mathit{ReceiveLeaseRequest}(i) \qquad \lor \mathit{ReceiveLeaseReply}(i)
\end{array}
$$

Figure 4: Initial condition and next-state relation specified in TLA$^+$.

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

Figure 5: Sending and receiving join requests.

particular, message loss does not have to be modeled by an explicit action because it is indistinguishable from the corresponding receive action never being executed.

Figure 5 shows the TLA$^+$ definitions of the *Join* and *ReceiveJoinRequest* actions. Messages are represented as records with fields *destination*, indicating the receiver of the message, and *content*, which itself is a record whose *type* field identifies the kind of the message. Action $Join(i, j)$ models a currently Dead node i sending a join request message to a Pastry node j that it knows about. The status of node i turns to Waiting and the request message is added to the message pool. In TLA$^+$, functions are total over their domain (written DOMAIN f), function application is written using square brackets, and the expression $[f \text{ EXCEPT } ![x] = e]$ represents a function update. More precisely, it denotes the function g that is similar to f, except that $g[x] = e$ when $x \in \text{DOMAIN } g = \text{DOMAIN } f$.

Action $ReceiveJoinRequest(i)$ models node i receiving a join request sent by a node that it covers.[3] As explained previously, the predicate $ToJoin[i] = i$ expresses the condition that node i does not currently help another node join the network. The action describes how the join request is consumed and a join reply, containing node i's leaf set, is sent to the requester. Moreover, node i records the fact that it now helps the requesting node join the network, and updates its leaf set by adding the new node.

Although the TLA$^+$ specification is written in terms of a global view of the protocol state, an action modeling a transition executed by node i accesses only the i-th entries of arrays stored in variables, as well as the message pool—by retrieving a message sent to node i or by adding messages. It can thus be implemented as a local action of a process in a distributed system.

3.3. Comparison With Lu's Specification

Our specification was derived from Lu's original specification of his variant of the Pastry algorithm [8], and it models the same algorithm. In order for the proof to gain in modularity, readability, and simplicity, we introduced additional operators such as *ClockwiseArc* or *ClosestFromTheRight* shown in Section 3.1. Some of these operators abstract from arithmetic calculations, others encapsulate the use of TLA$^+$'s CHOOSE operator, which is difficult for back-end provers to reason about, hampering automation.

We once and for all prove useful properties of these operators, eliminating the need for expanding their definitions in subsequent proofs. The following are examples of properties of predicate *ClockwiseArc*.

$$\begin{aligned}&\text{THEOREM } ArcReflexivity \triangleq \forall x, y \in I : \\ &\quad ClockwiseArc(x, y, y) \wedge ClockwiseArc(x, x, y)\end{aligned}$$

$$\begin{aligned}&\text{THEOREM } ArcAntisymmetry \triangleq \forall x, y, z \in I : \\ &\quad \wedge\ ClockwiseArc(x, y, z) \wedge ClockwiseArc(x, z, y) \Rightarrow y = z \\ &\quad \wedge\ ClockwiseArc(x, y, z) \wedge ClockwiseArc(y, x, z) \Rightarrow x = y\end{aligned}$$

[3] There is a similar action *RouteJoinRequest* modeling a node receiving a join request sent by a node that it does not cover, and which it forwards to a suitable node.

THEOREM $ArcRotation \triangleq \forall x, y, z \in I :$
$\quad \wedge\ x \neq y \wedge ClockwiseArc(x, y, z) \Rightarrow ClockwiseArc(y, z, x)$
$\quad \wedge\ y \neq z \wedge ClockwiseArc(x, y, z) \Rightarrow ClockwiseArc(z, x, y)$

These theorems are proved automatically using the SMT backend of TLAPS, and they can be used subsequently by backends such as Zenon or SPASS that do not have native support for integer arithmetic but that offer strong heuristics for quantifier instantiation.

For operators encapsulating CHOOSE expressions, we prove three elementary theorems. First, a *choose lemma* states the existence of a value satisfying the characteristic predicate. For our example, we prove

LEMMA $choose\_ClosestFromTheRight \triangleq \forall x \in I,\ S \in \text{SUBSET } I :$
$\quad S \neq \{\} \Rightarrow \exists y \in S : \forall z \in S : ClockwiseArc(x, y, z)$

by expanding the definition of *ClockwiseArc* and appealing to the existence of a smallest element in the set of natural numbers representing the clockwise distances between x and the elements of S. Next, we prove *type* and *expansion* lemmas that provide type information and state the characteristic properties of the operator. For this example, these lemmas are corollaries of the choose lemma, and their proofs are automatic. Appealing to these lemmas instead of expanding the operator definition avoids exposing the backends to CHOOSE expressions, which are notoriously difficult to handle. This method is generic; because CHOOSE returns an arbitrary value satisfying the predicate, the lemmas provide complete information about the defined operator.

LEMMA $type\_ClosestFromTheRight \triangleq \forall x \in I,\ S \in \text{SUBSET } I :$
$\quad ClosestFromTheRight(x, S) \in I$

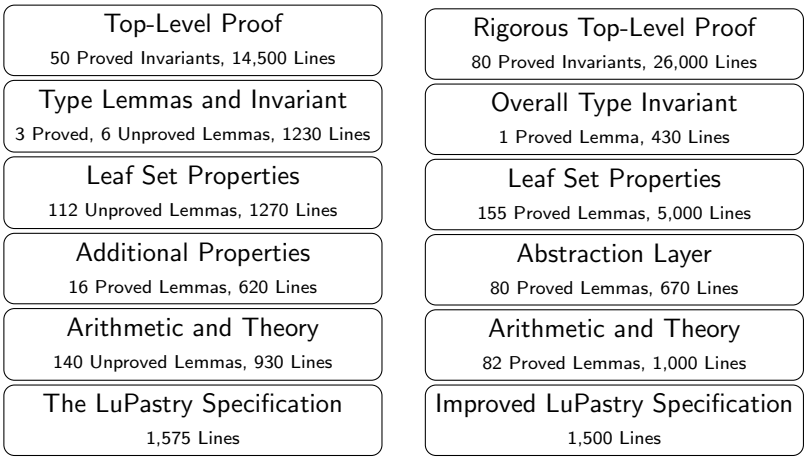
LEMMA $def\_ClosestFromTheRight \triangleq \forall x \in I,\ S \in \text{SUBSET } I :$
$\quad \wedge\ S = \{\} \Rightarrow ClosestFromTheRight(x, S) = x$
$\quad \wedge\ S \neq \{\} \Rightarrow ClosestFromTheRight(x, S) \in S$
$\quad \wedge\ \forall y \in S : ClockwiseArc(x, ClosestFromTheRight(x, S), y)$

We illustrate the effect of these abstractions using a simple lemma about adding new nodes to the leaf set data structure, that we prove once with and once without the use of the new operators.[4]

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

The proof of this lemma according to the original definition of *IsProper*, which did not use the predicate *ClockwiseArc*, consists of 23 interactive proof steps that generate 64 proof obligations. With our abstractions, the new proof consists of only 12 interactive proof steps (40 proof obligations). This significant difference comes from the fact that the new operators allow back-end provers to succeed directly on some steps in the new proof, which have to be broken down into further substeps in the original proof. Already for this simple example we observe a 50% reduction in the number of user interactions. At the end of Section 4.3 we further discuss proof automation from a more global perspective.

[4]See Section 3.1 for the definitions of *LeafSet*, *IsProper*, and *AddToLS*.



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

Figure 6: Overview of the proofs of LuPastry and LuPastry$^+$.

Besides the new abstractions, we fix some corner cases in the original specification. We modify the probing process so that a node does not probe itself. This is clearly unnecessary, and removing it simplifies some parts of the proof. We also add a missing border case to the definition of the operator $\mathit{FindNext}(i, j)$ that computes the next hop on the route from node i to node j based on the length of the shared prefix of digits between i and j.

4. Proving Pastry Correct

4.1. Overview of the Proof

Figure 6 summarizes the structure of our correctness proof and contrasts it with Lu's proof. It can be seen that Lu's proof contained a significant number of lemmas for which no proof was provided. They should rather be qualified as assumptions, in particular concerning arithmetic and the leaf set structure. Our proof consists of lemmas about arithmetic operators, the additional operators such as *ClockwiseArc*, and properties of the leaf set data structure, described in Section 4.2. The reasoning about the Pastry algorithm starts with proving type correctness and then establishes 80 invariants of the algorithm itself. Section 4.3 explains how overall correctness is proved as a consequence of the invariants.

4.2. Lemmas on the Data Structures

When examining the unproved lemmas at the bottom layers of Lu's proof, we found counter-examples to many of them, such as arithmetic assumptions ignoring border cases. Besides, several assumptions were not actually used in the proof. For example, Lu's proof relied on 112 unproved assumptions about the leaf set data structure. Upon examining these assumptions, we could prove only 21 directly. We discovered that more than 30 were unused in Lu's proof. The rest of the assumptions were incorrect. Our analysis of Lu's assumptions led us to reformulate those that were needed for the top-level proof. This was possible for all but 6 of the incorrect assumptions. For example, the following

assumption used by Lu states that after adding some set of nodes S to a leaf set $ls1$, the right neighbor of the resulting leaf set can only be closer to the leaf set owner than the original right neighbor of $ls1$.

$$
\begin{array}{l}
\text{LEMMA } \forall ls1 \in LeafSet,\ S \in \text{SUBSET } I : \\
\quad \text{LET } nd \triangleq ls1.node \\
\qquad ls2 \triangleq AddToLS(S, ls1) \\
\quad \text{IN } \ ClockwiseDistance(nd, RightNeighbor(ls2)) \\
\qquad \leq ClockwiseDistance(nd, RightNeighbor(ls1))
\end{array}
$$

This lemma does not hold if the right-hand part of the leaf set is empty (*i.e.*, $ls1.right = \{\}$), because in this case $RightNeighbor(ls1) = ls1.node$, which is closer to itself than to any other node. The lemma was therefore reformulated as follows.

$$
\begin{array}{l}
\text{LEMMA } \forall ls1 \in LeafSet,\ S \in \text{SUBSET } I : \\
\quad ls1.right \neq \{\} \Rightarrow \\
\quad \text{LET } nd \triangleq ls1.node \\
\qquad ls2 \triangleq AddToLS(S, ls1) \\
\quad \text{IN } \ ClockwiseDistance(nd, RightNeighbor(ls2)) \\
\qquad \leq ClockwiseDistance(nd, RightNeighbor(ls1))
\end{array}
$$

Other assumptions required deeper changes or had to be eliminated entirely. For example,

$$
\begin{array}{l}
\text{LEMMA } \forall ls \in LeafSet,\ k \in I : \\
\quad LeafSetContent(AddToLs(\{k\}, ls)) \setminus \{k\} = LeafSetContent(ls)
\end{array}
$$

claims that the leaf set obtained by adding a node k to some leaf set ls, contains the same nodes in ls, and possibly also k. This is not true: if the appropriate side of the leaf set already contains L elements, adding a new node k to it will generally result in some other node being removed from the leaf set, invalidating the claimed equality.

Besides reformulating and proving some assumptions from the original proof, we also added and proved new facts that were helpful for the proof, resulting in more lemmas in the "Leaf Set Properties" layer.

As a result of reformulating or eliminating assumptions from the lower levels of the proof, corresponding changes were required in higher-level proofs that relied on these assumptions. Worse, we found that the following claimed invariant of the protocol actually does not hold:

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

The predicate asserts that after some node n sent a *JoinReply* message, n's current neighbors can only be closer to it than its neighbors were at the time

when the message was sent. This is not true if n's leaf set was empty at the time the message was sent. As mentioned earlier, in case of an empty leaf set, the left and right neighbors of node n are n itself. Any new neighbors of n will have a larger distance from n than n itself. As a result of reconsidering the existing proof, we were led to introducing different invariants that underly our correctness proof (cf. the top-level box of Figure 6).

4.3. Proving Overall Correctness

Our main theorem proves correct delivery, as expressed by the following predicate [8], to hold throughout any execution of Pastry.

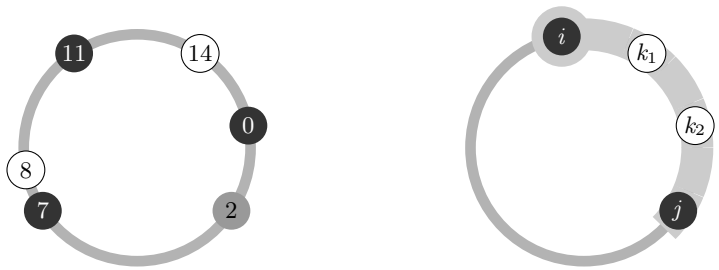
$$\begin{array}{l} \mathit{CorrectDelivery} \triangleq \forall i, k \in I : \text{ENABLED } \mathit{DeliverLookup}(i,k) \Rightarrow \\ \quad \wedge\ \forall n \in I : \mathit{Status}[n] = \text{“Ready”} \Rightarrow \\ \qquad\quad \mathit{AbsoluteDistance}(i,k) \leq \mathit{AbsoluteDistance}(n,k) \\ \quad \wedge\ \forall j \in I \setminus \{i\} : \neg\text{ENABLED } \mathit{DeliverLookup}(j,k) \end{array}$$

If $\mathcal{A}$ is an action formula, the TLA$^+$ predicate ENABLED $\mathcal{A}$ holds in those states s for which there exists some state t such that $\mathcal{A}$ holds over the pair $\langle s, t\rangle$. Formally, it is defined by quantifying existentially over the primed state variables that appear in $\mathcal{A}$. Since TLAPS currently does not support reasoning about ENABLED, Lu's proof as well as ours uses a reformulation of *CorrectDelivery* where the enabling predicate is unfolded manually.

The property requires that whenever node i can handle a lookup request for key k, then (a) no Ready node lies closer to k in the ring than i and (b) no other node can handle a lookup request for k. (The second condition serves to break a possible tie between two nodes with minimal distance from the key.) The action *DeliverLookup* is defined as follows.

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

We now outline the idea of the proof, using shorthand notation for some of the operator symbols. In particular, $RN(i) = \mathit{RightNeighbor}(\mathit{LeafSets}[i])$ denotes the right neighbor of node i as computed from the leaf set information, whereas $CR(i) = \mathit{ClosestFromTheRight}(i, \mathit{ReadyOKNodes} \setminus \{i\})$ is the closest Ready or OK node to the right of node i, based on the nodes actually present in the ring, independently of the nodes' knowledge. $LN(i)$ and $CL(i)$ similarly denote the left neighbor and the closest Ready or OK node to the left. We write $i_1 \rightharpoonup \cdots \rightharpoonup i_n$ to denote a clockwise path on the ring; this is the extension of the TLA$^+$ operator *ClockwiseArc* to an arbitrary number of nodes. The path between nodes i and j with the shortest absolute distance may be $i \rightharpoonup j$ or $j \rightharpoonup i$; we denote this shortest path by $i \rightleftharpoons j$. In a ring of 16 nodes, for example, $(3 \rightleftharpoons 5) = (3 \rightharpoonup 5)$, but $(3 \rightleftharpoons 15) = (15 \rightharpoonup 3)$. We write $|p|$ for the length of the path p.



(a) A ring with some nodes joining. (b) Ready and participating nodes.

Figure 7: Network Stability.

The proof relies on the fact that the coverage intervals of Ready nodes do not overlap, as explained in Section 2.1. The following predicate formalizes this idea by asserting that the clockwise distance from any Ready node n_1 to the right end point of its coverage interval is shorter than the clockwise distance from n_1 to the left end point of the interval for any other Ready node n_2.

$$\begin{aligned} &\mathit{NonOverlappingCoverage} \triangleq \forall n1, n2 \in \mathit{ReadyNodes} : n1 \neq n2 \Rightarrow \\ &\quad \mathit{ClockwiseDistance}(n1, \mathit{RightCoverage}(\mathit{LeafSets}[n1])) \\ &\quad < \mathit{ClockwiseDistance}(n1, \mathit{LeftCoverage}(\mathit{LeafSets}[n2])) \end{aligned}$$

The proof of non-overlapping coverage relies on the following property, which (adapting notation) was already used by Lu [8] as a main invariant.

$$\begin{aligned} &\mathit{CloseNeighbors} \triangleq \forall n1, n2 \in \mathit{ReadyNodes} : n1 \neq n2 \Rightarrow \\ &\quad \wedge\ \mathit{ClockwiseArc}(n1, \mathit{RN}(n1), n2) \\ &\quad \wedge\ \mathit{ClockwiseArc}(n2, \mathit{LN}(n1), n1) \end{aligned}$$

Lu's partial proof of correct delivery relies on the property *CloseNeighbors*, which is in turn proven mainly by reasoning about the lease exchange phase of the join protocol. Instead, we observe that *CloseNeighbors* is a consequence of a stronger property that we call *stable network*, and which we prove by reasoning mainly about the probing phase of the join protocol, rather than the lease exchange phase.

A node i is *stable* if the Ready or OK nodes closest to it on either side, i.e., $CR(i)$ and $CL(i)$, are members of i's leaf set. A Pastry ring is *stable* if all Ready or OK nodes are stable. For example, the ring shown in Figure 7a is stable if nodes 0, 2, 7, and 11 are stable. For node 0 to be stable, it has to contain node 11 in its left leaf set and node 2 in its right leaf set. (Node 14 is Waiting, so node 0 does not yet need to know about it.) It is easy to see that the properties *CloseNeighbors*, and therefore also *NonOverlappingCoverage*, hold in a stable ring.

We prove that when the leaf set size L is at least 3, the Pastry ring is always stable. Let a *participating node* be a node that is either Ready or OK, or that is the to-join node of a Ready node. Essentially, a participating node is any node that is known to some Ready or OK node. Let i and j be two consecutive Ready or OK nodes on the ring (see Figure 7b). There can be at most two participating nodes k_1, k_2 between i and j: the to-join nodes of i and j. Any other non-Dead node between i and j must be a Waiting node whose join request has not been

picked up by i or j (since they are busy facilitating the joins of k_1 and k_2), and so it can not be in the leaf sets of i or j. For a leaf set size $L \geq 3$, we can ensure that stable nodes i and j remain stable even if new nodes are added to their leaf sets.

We have proved in TLA$^+$ the following invariants of the Pastry ring.

P1. The coverage of a node is computed based on half the distance to its neighbors. A key k covered by a node i lies in either the *right* or *left* coverage regions of i (see Figure 1). If i and j are leaf set neighbors, *i.e.*, $i = LN(j)$ and $j = RN(i)$, their coverage regions cannot overlap.

P2. If k is in i's right (resp., left) coverage region, then $i \rightharpoonup k \rightharpoonup RN(i)$ (resp., $LN(i) \rightharpoonup k \rightharpoonup i$).

P3. If i is a stable node and $r \neq i$ is some Ready or OK node, then $i \rightharpoonup RN(i) \rightharpoonup r$ and $r \rightharpoonup LN(i) \rightharpoonup i$. Assuming $L \geq 3$, there is always room in node i's leaf set for i's Ready/OK neighbor j (see Figure 7b).

P4. Because we exclude node failure, all protocol actions that modify a node's leaf set do so through the operation *AddToLS*. Therefore, nodes are not purposely removed from a leaf set, but a node j may only be evicted from the leaf set for node i through an *AddToLS* operation that results in an overflow; i.e., if the leaf set of i becomes full and j is replaced by another node that is closer to i.

P5. A new node k joins the network through a Ready node i that initially covers it, and so k will remain closest to i on one side (right or left) until it finishes its join process. Only after k has finished joining and turned Ready, other nodes can join the network between k and i. Therefore, any participating node between i and $CR(i)$ is either $ToJoin[i]$ or $ToJoin[CR(i)]$. That is, there can never be three different participating nodes k_1, k_2, k_3 such that $i \rightharpoonup k_1 \rightharpoonup k_2 \rightharpoonup k_3 \rightharpoonup CR(i)$, or dually, $CL(i) \rightharpoonup k_1 \rightharpoonup k_2 \rightharpoonup k_3 \rightharpoonup i$.

P6. Assuming $L \geq 3$, no action can cause $CR(i)$ or $CL(i)$ to be removed from i's leaf set due to an overflow (see Figure 7b).

P7. At any point in time, a participating node i is either probing $CR(i)$ or $CR(i)$ is in i's leaf set, and similarly for $CL(i)$.

P8. At any point in time, a participating node i is either probing $CR(i)$ or i is in the leaf set of $CR(i)$, and similarly for $CL(i)$.

Based on these invariants, the proof of correctness is subdivided into the following two theorems.

Theorem 1. *CorrectDelivery holds in any stable Pastry ring.*

Proof. The action *DeliverLookup*(i, k) is enabled only if i is a Ready node that covers key k. Assume for the sake of contradiction that *DeliverLookup*(i, k) and *DeliverLookup*(j, k) are enabled for two different nodes $j \neq i$. Nodes i and j are Ready, and both cover k. W.l.o.g., assume that k is in i's right coverage region and in j's left coverage region, hence $i \rightharpoonup k \rightharpoonup j$. By P2 and P3, we have $i \rightharpoonup k \rightharpoonup RN(i) \rightharpoonup j$ and $i \rightharpoonup LN(j) \rightharpoonup k \rightharpoonup j$. In order for both to hold, it must be that $i = LN(j)$ and $j = RN(i)$. Now, P1 implies that the coverage regions of i and j do not overlap. Therefore, in a stable Pastry ring, two Ready nodes i and j cannot both cover the same key k, and at most one of *DeliverLookup*(i, k) and *DeliverLookup*(j, k) is enabled.

It remains to show that if *DeliverLookup*(i, k) is enabled, then i is closer to k than any other Ready node r is in terms of absolute distance on the ring. Assume again that k is in i's right coverage region, and so $i \rightharpoonup k \rightharpoonup RN(i) \rightharpoonup r$ (by P3). Because k lies within half the distance from i to $RN(i)$ (by P1), k must lie within half the distance from i to r. Therefore, $|i \rightharpoonup k| = |i \rightleftharpoons k| \leq |k \rightharpoonup r|$. If $|r \rightleftharpoons k| = |k \rightharpoonup r|$, we are done. Otherwise, we have $|r \rightleftharpoons k| = |r \rightharpoonup k|$. Because of the ordering of the nodes on the ring ($r \rightharpoonup i \rightharpoonup k$), we have that $|r \rightharpoonup k| = |r \rightharpoonup i| + |i \rightharpoonup k|$. Since path lengths are non-negative, it follows that $|i \rightleftharpoons k| \leq |r \rightleftharpoons k|$. □

Theorem 2. *For $L \geq 3$, the Pastry ring is always stable.*

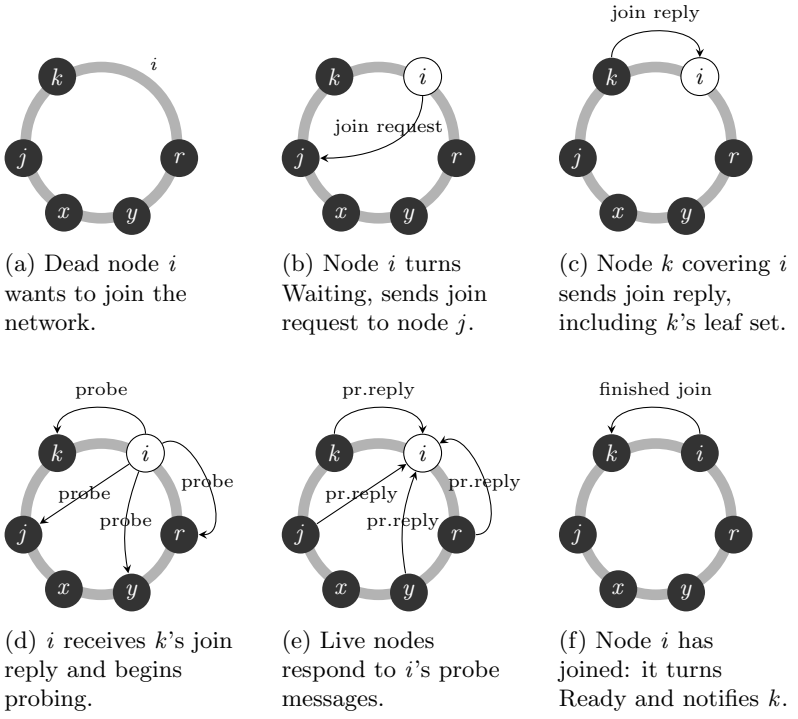
Proof. The definition of *Init* implies that the Pastry ring is stable in the initial state: nodes in A are Ready and all other nodes are Dead. The leaf set of each node $i \in A$ is initialized by adding the L closest nodes among A on either side to i's empty leaf set. Consequently, the leaf set of node i contains its closest right and left neighbors in A. All A-nodes are stable, and so the network is stable.

Now consider a stable Pastry ring R. For the induction step, we need to show that R remains stable after executing any possible transition. We denote by R' the state of the ring after the transition, and write $CR'(i)$ and $CL'(i)$ to denote the values of $CR(i)$ and $CL(i)$ of node i in R'. We need to show that (a) if the considered transition results in some unstable node i in R turning Ready or OK, then i is stable in R', and (b) all stable nodes in R remain stable in R'.

- (a) Let i be an unstable node in R. Since R is stable, i is not Ready or OK. The only action that can change i into a Ready or OK node in R' is *ReceiveProbeReply*(i): i receives the last probe reply message it was waiting for, its probing set becomes empty, and i becomes OK. Since that transition only changes the local variables of i, the status of all other nodes remains unchanged; $CR'(i) = CR(i)$ and $CL'(i) = CL(i)$. Using P7, and since i's probing set is empty, $CR(i)$ and $CL(i)$ are in i's leaf set, hence i is stable in R'.
- (b) Assume that i is a stable node in R. If $CR'(i) = CR(i)$ and $CL'(i) = CL(i)$, then i remains stable in R' by P6. Now suppose, w.l.o.g., that $CR'(i) \neq CR(i)$. The only transition causing a change of the closest Ready/OK node to the right is *ReceiveProbeReply*(j), where $CR'(i) = j$. Now, $i = CL'(j)$. By P8, and since node j's probing set is empty, j is in i's leaf set. Therefore, i remains stable. □

Our machine-checked TLA$^+$ proof consists of more than 30,000 lines. It relies on 80 invariants, whose proofs amount to over 20,000 lines overall, whereas 10,000 lines of proof are required for the underlying lemmas and the main correctness theorems. It took around two person-years to develop the proof (including mastering TLAPS), with the main effort being the design of the invariants. The time taken to run TLAPS on the entire proof is 8 hours and 57 minutes on a single Intel Xeon(R) CPU E5-2680 core running at 2.7 GHz with 256 GB RAM.

The maximum nesting depth of our proof is 6, with an average nesting depth of just below 3. This contrasts favorably with Lu's proof, where the maximum nesting depth is at least 8, indicating that our proof benefits from significantly



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

(b) Node i turns Waiting, sends join request to node j.

(c) Node k covering i sends join reply, including k's leaf set.

(d) i receives k's join reply and begins probing.

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

(f) Node i has joined: it turns Ready and notifies k.

Figure 8: The Simplified Join Protocol.

more automation. Unfortunately, it is not straightforward to compare the two proofs. First, in Lu's proof the manually declared depth of many steps is higher than the actual nesting depth. Second, our proof has a different top-level structure. Lu's proof considers each action in turn, and then proves every invariant, whereas we have separate proofs for each invariant. Still, we attribute the improvement on maximum and average nesting depth to the introduction of the new operators providing a new layer for abstracting arithmetic ring properties and our introduction of specific lemmas about operators defined in terms of CHOOSE expressions.

5. Simplifying the Join Protocol

Analyzing the correctness proof outlined in Section 4, we notice that it is independent of the lease exchange phase of the join protocol. In what follows, we simplify the join process of Pastry by eliminating the lease exchange phase, while still maintaining lookup consistency. We denote the simplified specification by Simplified LuPastry$^+$, or Simplified Pastry.

5.1. The Simplified Pastry Join Process

Our simplified join process is illustrated in Figure 8, and it starts in the same manner as the original process shown in Figure 2. A Dead node i that decides to join the network turns to Waiting and sends a *join request* to a Ready node j that it knows about. The request is forwarded to the Ready node k that covers

node i. We call k the node *responsible* for i. Node k responds to i's request when it is free for handling a new join request. Node i receives k's reply and starts the probing phase, during which leaf sets of the participating nodes are updated as explained in Section 2.1.

The simplified protocol differs from the original one at the end of the probing phase. In the simplified protocol, node i turns Ready as soon as it finishes the probing phase. Instead of sending a lease request, it notifies k that it has finished the join process (see Figure 8f). Once k has received i's notification, it may help other new nodes join the network. Messages relating to leases, as well as the OK status, have been removed from the protocol.

The outline of the TLA$^+$ specification of the simplified protocol appears in Figure 9; it is very similar to our original specification from Section 3. The global state of the network is represented as the tuple *svars*. One new variable is introduced: $Responsible[i]$ denotes the Ready node responsible for i's join (if any), otherwise, $Responsible[i] = i$. This variable is needed for node i, at the end of the join protocol, to notify the node that helped it join. In the previous version, that notification was implicit in the lease request message sent after the joining node turned OK. On the other hand, we omit the variables *Leases* and *Grants*, since they are no longer needed in the simplified protocol.

The definitions of the initial state predicate and next-state relation of the simplified protocol are derived by replacing lease exchange actions by the new action *ReceiveNotification*, where $ReceiveNotification(i)$ models node i receiving a notification from the node it is responsible for about the end of the join protocol. Node i resets its $ToJoin[i]$ field back to i so that it can help new nodes join the network. The overall TLA$^+$ specification of the simplified protocol is defined as

$$SSpec \triangleq SInit \wedge \Box[SNext]_{svars}$$

Because nodes never turn OK in the simplified protocol, it is always the case that $OKNodes = \{\}$, and $ReadyOKNodes = ReadyNodes$.

5.2. Proving Correctness for Simplified Pastry

Adapting the proof of correct delivery for the original specification to the simplified version is straightforward, despite the slight modifications in the protocol specification. The proof described in Section 4 mainly focuses on the probing process and proves that the nodes in a LuPastry$^+$ network that are Ready or OK are always stable. In particular, every Waiting node must probe or be probed by its left and right Ready/OK neighbors before it turns OK. That is, the probing phase already establishes and maintains stability for both Ready and OK nodes, rather than just the Ready nodes. Eliminating the lease exchange phase and the intermediate OK status effectively reduces the set of Ready/OK nodes to the set of Ready nodes. For the simplified protocol, the probing phase therefore establishes that the Ready nodes are always stable.

In order to complete the proof, it is enough to notice that exclusive coverage and correct delivery follow from stability, and that this implication does not depend on the lease exchange phase. Adapting the formal TLA$^+$ proof required the following steps.

1. *Removing invariants.* Invariants relating to the lease exchange phase are no longer needed and are therefore eliminated. These are invariants relating to either exchanged lease request and reply messages, or the variables

$$svars \triangleq \langle Messages, Status, LeafSets, Probing, ToJoin, Responsible \rangle$$

$$\begin{aligned}
&SInit \triangleq \\
&\quad \land\ Messages = \{\} \\
&\quad \land\ Status = [i \in I \mapsto \text{IF } i \in A \text{ THEN } \text{“Ready”} \text{ ELSE } \text{“Dead”}] \\
&\quad \land\ LeafSets = [i \in I \mapsto \text{IF } i \in A \text{ THEN } AddToLS(A, EmptyLS(i)) \\
&\qquad\qquad\qquad\qquad\qquad \text{ELSE } EmptyLS(i)] \\
&\quad \land\ Probing = [i \in I \mapsto \{\}] \\
&\quad \land\ ToJoin = [i \in I \mapsto i] \\
&\quad \land\ Responsible = [i \in I \mapsto i]
\end{aligned}$$

$$\begin{aligned}
&SNext \triangleq \exists i, j \in I : \\
&\quad \lor Lookup(i,j) && \lor RouteLookup(i,j) \\
&\quad \lor DeliverLookup(i,j) && \lor Join(i,j) \\
&\quad \lor RouteJoinRequest(i,j) && \lor ReceiveJoinRequest(i) \\
&\quad \lor ReceiveJoinReply(i) && \lor ReceiveProbe(i) \\
&\quad \lor ReceiveProbeReply(i) && \lor ReceiveNotification(i)
\end{aligned}$$

Figure 9: Specifying the simplified protocol.

Leases and *Grants*. The proof of other invariants is mostly independent from invariants on lease exchange, which only intervene during the final steps of the join protocol.

2. *Modifying invariants.* The proof of the remaining invariants has to be adjusted for the next-state relation of the simplified protocol. Induction steps for the eliminated actions *RequestLease*, *ReceiveLeaseRequest* and *ReceiveLeaseReply* are removed. For the new action *ReceiveNotification*, one induction step is added.
3. *Adding invariants.* A few very simple invariants are added in order to prove some properties about the new variable *Responsible*, such as the following invariant.

$$\begin{aligned}
&\forall i \in ToJoinNodes \setminus ReadyNodes : \\
&\qquad LeafSets[i] \neq EmptyLS(i) \Rightarrow i \neq Responsible[i]
\end{aligned}$$

The new proof is comparable in size to our original proof, both in terms of number of proof interactions and the number of invariants. However, once we realized that the lease exchange phase was hardly used in the proof, adapting the specification and the proofs was a matter of a few person-days, yielding a pleasant experience in reuse. One could have expected that the new proof would be significantly shorter than the original one, given that the protocol has been simplified. However, the original proof derived non-overlapping coverage from properties established by lease exchange, whereas the new proof required some more arithmetic reasoning steps in order to show that the coverage intervals determined by the leaf sets of Ready nodes do not overlap. The same arguments could have been used in the original proof, in which case the new proof would indeed be much smaller than the original one.

6. Conclusion

In this paper, we formally and rigorously proved correctness of the join protocol of a version of the Pastry algorithm in a pure-join model. We first analyzed Lu's existing correctness proof for LuPastry [8], mechanized in the TLA$^+$ proof assistant, that relied on many unproved assumptions. Several assumptions were found to contain errors, and we proved a set of lemmas about arithmetic and data structures that allowed us to give a full correctness proof of the protocol. Second, our TLA$^+$ specification of LuPastry introduces a layer of abstractions for arithmetic reasoning and for operators defined in terms of CHOOSE expressions, which help improve the degree of automation of the proof. Third, we present a simplified version of the join protocol, and adapt our first proof to prove correct lookups for the simplified version. In particular, we eliminate the *lease exchange* phase of the join process: a handshaking step between a new node and its neighbor nodes before the new node becomes an active participant in the Pastry ring. Lu showed that this lease exchange step is not enough for guaranteeing correct lookups in the case of the full protocol where nodes join and leave freely. We observe, on the other hand, that the pure-join model does not require this step.

Our proof relies on proving that the set of Ready nodes of a Pastry ring is stable, and proving stability requires that the size of the leaf set (on either side of the node) be at least 3. We believe that our main correctness property of correct delivery holds independently of this assumption (even for $L = 1$) in a pure-join model, but that stability is a useful property of Pastry in more general settings. Typical implementations of Pastry use values of $L = 8$ or $L = 16$, and therefore our assumption does not appear to be a restriction in practice.

It would be interesting to extend our proof to a version of Pastry where nodes can leave and/or fail. As in the case of other DHT protocols [12, 14], it is easy to see that the Pastry ring may become disconnected if too many nodes quit in a given neighborhood. The Pastry algorithm may be able to repair the leaf sets and ensure correct routing as long as not more than L consecutive nodes leave the ring, although stale messages from a node that failed during a previous join attempt and then attempts to reconnect may confuse the protocol. However, we have not yet been able to formally prove or disprove such a bound on the number of neighbor nodes that may sign off.

As an exercise in formal verification, our experience confirms that TLA$^+$ is well-suited for modeling concurrent and distributed algorithms such as Pastry. In particular, the set-theoretic nature of the specification language encourages writing a formal model of the algorithm at a suitably high level of abstraction. Moreover, TLA$^+$'s hierarchical proof language lets a user focus on parts of the proof without having to remember details about unrelated parts of the proof.

The possibility to run the TLC model checker on the same formal models used in the proof proved to be extremely useful. We used model checking to validate potential invariants before attempting to prove them. Quite often, the model checker would immediately disprove a property thought to be an invariant, even for small instances of the algorithm, and the counterexamples were extremely helpful in eventually finding the correct invariants to prove. Moreover, although we have not formally proved any liveness properties, the use of model checking helped us gain confidence that our specification of Pastry would ensure liveness of the protocol, given suitable fairness assumptions. For

example, we have seen in model checking rings of 4 and 8 nodes that it is not possible for the protocol to reach a deadlock until all nodes turn Ready. In particular, the protocol can always find a Ready node that covers a newly arriving node, and the probing phase is similarly carried out successfully.

Like Lu, we rely on a large invariant for proving the main safety property of the protocol "in one shot", rather than proceeding by refinement from a high-level model where nodes join atomically, down to a detailed model of the real protocol. TLA$^+$ has a notion of refinement based on trace inclusion, and it would be interesting to develop a refinement-based proof and compare its complexity to that of our proof. The difficulty is that parts of the state, such as contents of leaf sets under construction, become visible when some node completes its join protocol, revealing information about other nodes joining concurrently. This precludes obtaining our specification as a refinement of a high-level specification where new nodes join atomically. It is also not clear to us to what extent elements of our specification or proof could be generalized to related distributed algorithms.

On a technical level, TLAPS includes facilities for checking the status of a proof that can identify which steps are affected by a change in the specification or the proof. While TLAPS can manage a proof of the size reported here, it is barely able to do so. For example, the Java heap size allotted to Eclipse has to be increased to several gigabytes, and status checking currently takes almost as much time as rerunning the proof. Although users of a mechanical theorem prover always dream of better automation, the main difficulty in the formal verification of algorithms is in fact finding sufficiently strong inductive invariants that underpin the correctness argument, and any assistance in this task would be most welcome.

Acknowledgement. We are very grateful to the anonymous reviewers of both the ABZ paper [17] and a previous version of this article for their insightful comments and suggestions that helped us improve the exposition.

References

[1] G. H. L. Fletcher, H. A. Sheth, K. Börner, Unstructured peer-to-peer networks: Topological properties and search performance, in: G. Moro, S. Bergamaschi, K. Aberer (Eds.), Agents and Peer-to-Peer Computing (AP2PC 2004), Vol. 3601 of LNCS, Springer, New York, U.S.A., 2005, pp. 14–27.

[2] A. Rowstron, P. Druschel, Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems, in: R. Guerraoui (Ed.), Distributed Systems Platforms (Middleware'01), Vol. 2218 of LNCS, Springer, Heidelberg, Germany, 2001, pp. 329–350.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, H. Balakrishnan, Chord: A scalable peer-to-peer lookup service for Internet applications, in: R. L. Cruz, G. Varghese (Eds.), Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2001), ACM, San Diego, U.S.A., 2001, pp. 149–160.

[4] P. Maymounkov, D. Mazières, Kademlia: A Peer-to-Peer Information System Based on the XOR Metric, in: Peer-to-Peer Systems (IPTPS'01), Vol. 2429 of LNCS, Springer, Cambridge, U.S.A., 2002, pp. 53–65.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A Scalable Content-Addressable Network, in: R. L. Cruz, G. Varghese (Eds.), Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2001), ACM, San Diego, U.S.A., 2001, pp. 161–172.

[6] L. O. Alima, S. El-Ansary, P. Brand, S. Haridi, $DKS(N, k, f)$: A family of low communication, scalable and fault-tolerant infrastructures for p2p applications, in: Cluster Computing and the Grid (CCGrid 2003), IEEE Comp. Soc., Tokyo, Japan, 2003, pp. 344–350.

[7] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, H. Vanzetto, TLA$^+$ Proofs, in: D. Giannakopoulou, D. Méry (Eds.), Formal Methods (FM 2012), Vol. 7436 of LNCS, Springer, Paris, France, 2012, pp. 147–154.

[8] T. Lu, Formal verification of the Pastry protocol using TLA$^+$, in: X. Li, Z. Liu, W. Yi (Eds.), Dependable Software Engineering: Theories, Tools, and Applications (SETTA 2015), Vol. 9409 of LNCS, Springer, Nanjing, China, 2015, pp. 284–299.

[9] T. Lu, Formal Verification of the Pastry Protocol, Ph.D. thesis, Universität des Saarlandes and Université de Lorraine (2013).

[10] A. Haeberlen, J. Hoye, A. Mislove, P. Druschel, Consistent key mapping in structured overlays, Tech. Rep. TR05-456, Rice University, Department of Computer Science (2005).

[11] R. Bakhshi, D. Gurov, Verification of peer-to-peer algorithms: A case study, in: Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems (MTCoord 2006), Vol. 181 of ENTCS, Elsevier, 2007, pp. 49–57.

[12] P. Zave, Using Lightweight Modeling to Understand Chord, ACM SIGCOMM Computer Communication Review 42 (2) (2012) 49–57.

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

[14] A. Ghodsi, Distributed k-ary system: Algorithms for distributed hash tables, Ph.D. thesis, KTH Royal Institute of Technology, Stockholm, Sweden (2006).

[15] J. Borgström, U. Nestmann, L. O. Alima, D. Gurov, Verifying a structured peer-to-peer overlay network: The static case, in: Global Computing (GC 2004), Vol. 3267 of LNCS, Springer, Rovereto, Italy, 2004, pp. 250–265.

[16] T. Lu, S. Merz, C. Weidenbach, Towards Verification of the Pastry Protocol Using TLA$^+$, in: R. Bruni, J. Dingel (Eds.), Formal Techniques for Distributed Systems (FORTE 2011), Vol. 6722 of LNCS, Springer, Reykjavik, Iceland, 2011, pp. 244–258.

[17] N. Azmy, S. Merz, C. Weidenbach, A Rigorous Correctness Proof for Pastry, in: M. J. Butler, K.-D. Schewe, A. Mashkoor, M. Biró (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016), Vol. 9675 of LNCS, Springer, Linz, Austria, 2016, pp. 86–101.

[18] L. Lamport, Specifying Systems, Addison-Wesley, Boston, Mass., 2002.

[19] Y. Yu, P. Manolios, L. Lamport, Model checking TLA$^+$ Specifications, in: L. Pierre, T. Kropf (Eds.), Correct Hardware Design and Verification Methods (CHARME'99), Vol. 1703 of LNCS, 1999, pp. 54–66.

[20] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, C. Tinelli, CVC4, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification (CAV'11), Vol. 6806 of LNCS, Springer, Snowbird, UT, 2011, pp. 171–177.
URL `http://dl.acm.org/citation.cfm?id=2032305.2032319`

[21] T. Bouton, D. Caminha de Oliveira, D. Déharbe, P. Fontaine, veriT: An open, trustable and efficient smt-solver, in: R. Schmidt (Ed.), Automated Deduction (CADE-22), Vol. 5663 of LNCS, Springer, Montreal, Canada, 2009, pp. 151–156.

[22] L. M. de Moura, N. Bjørner, Z3: an efficient SMT solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008), Vol. 4963 of LNCS, Springer, Budapest, Hungary, 2008, pp. 337–340.

[23] C. Weidenbach, D. Dimova, A. Fietzke, M. Suda, P. Wischnewski, SPASS version 3.5, in: R. Schmidt (Ed.), Automated Deduction (CADE-22), Vol. 5663 of LNCS, Springer, Montreal, Canada, 2009, pp. 140–145.

[24] R. Bonichon, D. Delahaye, D. Doligez, Zenon: An extensible automated theorem prover producing checkable proofs, in: N. Dershowitz, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2007), Vol. 4790 of LNCS, Springer, Yerevan, Armenia, 2007, pp. 151–165.

[25] L. C. Paulson, Isabelle: A Generic Theorem Prover, Vol. 828 of Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg, 1994.

[26] M. Suda, C. Weidenbach, A PLTL-prover based on labelled superposition with partial model guidance, in: B. Gramlich, D. Miller, U. Sattler (Eds.), Automated Reasoning (IJCAR 2012), Vol. 7364 of LNCS, Springer, Manchester, UK, 2012, pp. 537–543.