Towards Verification of the Pastry Protocol using TLA$^+$

Tianxiang Lu Stephan Merz
Christoph Weidenbach

MPI–I–2011–MPI-I-2011-RG1-002
June 2011

Authors' Addresses

Christoph Weidenbach
Max-Planck-Institut für Informatik
Campus E 1 4

D-66123 Saarbrücken

Stephan Merz
INRIA Nancy & LORIA
615, rue du Jardin Botanique

F-54602 Villers-lès-Nancy, France

Tianxiang Lu
Max-Planck-Institut für Informatik
Campus E 1 4

D-66123 Saarbrücken

Publication Notes

This report is an extended version of an article published elsewhere.

Abstract

Pastry is an algorithm that provides a scalable distributed hash table over an underlying P2P network. 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* and fault tolerance, it makes an interesting target for verification. We have modeled Pastry's core routing algorithms and communication protocol in the specification language TLA$^+$. In order to validate the model and to search for bugs we employed the TLA$^+$ model checker TLC to analyze several qualitative properties. We obtained non-trivial insights in the behavior of Pastry through the model checking analysis. Furthermore, we started to verify Pastry using the very same model and the interactive theorem prover TLAPS for TLA$^+$. A first result is the reduction of global Pastry correctness properties to invariants of the underlying data structures.

Keywords

formal specification, model checking, verification methods, network protocols

Contents

1 Introduction

Pastry [15, 4, 8] is an overlay network protocol that implements a distributed hash table. The network 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, i.e., $2^M - 1$ 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. An active node is in particular responsible for keys that are numerically close to its network 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.

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 *Correct Key Delivery*, requiring that there is always at most one node responsible for a given key. This property is non-trivial to obtain in the presence of spontaneous arrival and departure of nodes. Nodes may simply drop off, and Pastry is meant 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 set neighbors. If a node detects, e.g. by a ping, that one of its direct neighbor nodes dropped off, the node takes actions to recover from this state. So the value of l is relevant for the amount of "drop off" and fault tolerance of the protocol.

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 logarithmic number of steps in the size of the ring.

Pastry routes a message by forwarding it to nodes that match progres-

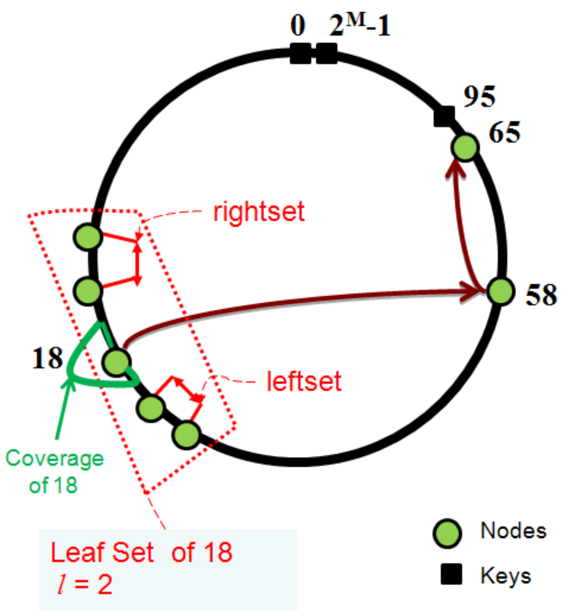


Figure 1.1: Pastry Routing Example

sively longer prefixes with the destination key. In the example of Figure 1.1, node 18 received a lookup message for key 95. The key is outside node 18'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 18 finds node 58, whose identifier matches the longest prefix with the destination key and then forwards the message to that node. Node 58 repeats the process and finally, the lookup message is answered by node 65, which is the closest node to the key 95, i.e., it covers key 95. In this case, we say that node 65 *delivers* the lookup request for key 95 (see also Figure 3.2).

The first challenge in modeling Pastry was to determine an appropriate level of abstraction. As a guiding principle, we focused the model toward supporting detailed proofs of the correctness properties. We abstracted from an explicit notion of time because it does not contribute to the verification of correctness properties. For example, time-triggered periodic maintenance messages exchanged between neighbors are modeled by non-deterministic sending of such messages. In contrast, we developed a detailed model for the address ring, the routing tables, the leaf sets, as well as the messages and actions of the protocol because these parts are central to the correctness of Pastry.

The second challenge was to fill in needed details for the formal model that are not contained in the published descriptions of Pastry. Model checking was very helpful for justifying our decisions. For instance, it was not explicitly stated what it means for a leaf set to be "complete", i.e., when a node starts taking over coverage and becoming an active member on the ring. It was not stated whether an overlap between the right and left leaf set is permitted or whether the sets should always be disjoint. We made explicit assumptions on how such corner cases should be handled, sometimes based on an exploration of the source code of the FreePastry implementation [13]. Thus, we implemented an overlap in our model only if there are at most $2l$ nodes present on the entire network, where l is the size of each leaf set. A complete leaf set only contains less than l nodes, if there are less than l nodes on the overall ring.

A further challenge was to formulate the correctness property; in fact, it is not stated explicitly in the literature [15, 4, 8]. The main property that we are interested in is that the lookup message for a particular key is answered by at most one "ready" node covering the key. We introduced a more fine-grained status notion for nodes, where only "ready" nodes answer lookup and join requests. The additional status of a node being "ok" was added in the refined model described in Section 4 to support several nodes joining simultaneously between two consecutive "ready" nodes.

The report is organized as follows. In Section 2 we explain the basic mechanisms behind the join protocol of Pastry. This protocol is the most important part of Pastry for correctness. Key aspects of our formal model are introduced in Section 3. Here we show the TLA$^+$ formal model at the corresponding places where we introduce them in the text. To the best of our knowledge, we present the first formal model covering the full Pastry algorithm. A number of important properties are model checked, subsections 3.3–3.4 and subsections 4.2–4.3, and the results are used to refine our model in case the model checker found undesired behavior. In addition to model checking our model, we have also been able to prove an important reduction property of Pastry. Basically, the correctness of the protocol can be reduced to the consistency of leaf sets, as we show in Section 5, Theorem 3. The report ends with a summary of our results, related work, and future directions of research in Section 6.

2 The Join Protocol

The most sophisticated part of Pastry is the protocol for a node to join the ring. In its simplest form, a single node joins between two "ready" nodes on the ring. The new node receives its leaf sets from the "ready" nodes, negotiates with both the new leaf sets and then goes to status "ready".

The join protocol is complicated because any node may drop off at any time, in particular while it handles a join request. Moreover, several nodes may join the ring concurrently between two adjacent "ready" nodes. Still, all "ready" nodes must agree on key coverage.

Figure 2.1 presents a first version of the join protocol in more detail, according to our understanding from [15] and [4]. We will refine this protocol in Sect. 4 according to the description in [8].

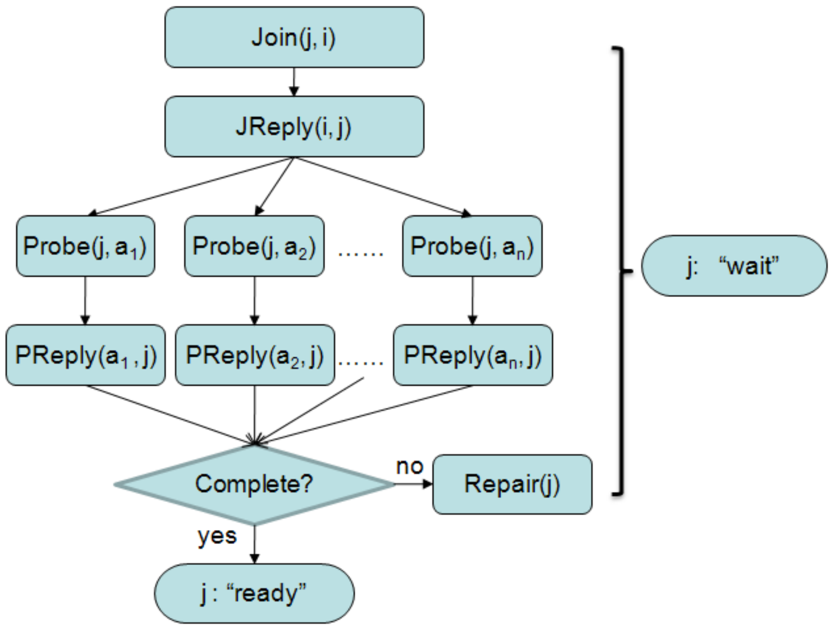


Figure 2.1: Overview of the join protocol.

Node j announces its interest in joining the ring by performing a *Join* action. At this point, its status is “wait”. Its join request will be routed to the closest “ready” node i just like routing a lookup message, treating j as the key. Node i replies to j by performing a join reply, *JReply* action, transmitting its current leaf sets to enable node j to construct its own leaf sets. Then the node j *probes* all nodes in its leaf sets in order to confirm their presence on the ring. A probe reply, action *PReply*, signals j that the respective leaf set node received the probe message from j and updated its local leaf set with j. The reply contains the updated leaf set. Each time the node j receives a probe reply 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 set is *complete*. If it is, it finishes the join phase and goes to status “ready”. Otherwise, any fault case is summarized in Figure 2.1 by *Repair*. For example, if a probe eventually fails, the probed node needs to be removed from the leaf set. Then the node j probes the most distant nodes (leftmost and rightmost) in its leaf sets to get more nodes, retrying to complete its leaf set.

3 A First Formal Model of Pastry

We modeled Pastry as a (potentially infinite-state) transition system in TLA$^+$ [9]. Although there are of course alternative logics and respective theorem provers for modeling Pastry, TLA$^+$ fits protocol verification quite nicely, because its concept of actions matches the rule/message based definition of protocols. Our model is available on the Web[1]. We explain those parts of the model that are used later on for model checking and for proving the reduction theorem.

3.1 Static Model

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. The following definition introduces different notions of distances between nodes or keys that will be used in the model.

Definition 1 (Distances). *Given x, y* $\in I$*:*

$$\begin{aligned}
\mathit{Dist}(x, y) &\triangleq \begin{cases} x - y + 2^{M-1} & \mathit{if} \;\; x - y < -2^{M-1} \\ x - y - 2^{M-1} & \mathit{if} \;\; x - y > 2^{M-1} \\ x - y, & \mathit{else} \end{cases} \\
\mathit{AbsDist}(x, y) &\triangleq |\mathit{Dist}(x, y)| \\
\mathit{CwDist}(x, y) &\triangleq \begin{cases} \mathit{AbsDist}(x, y) & \mathit{if} \;\; \mathit{Dist}(x, y) < 0 \\ 2^M - \mathit{AbsDist}(x, y) & \mathit{else} \end{cases}
\end{aligned}$$

[1]http://www.mpi-inf.mpg.de/~tianlu/software/PastryModelChecking.zip

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 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.

```
------------------------- MODULE Ring ------------------------
EXTENDS
   TLC,       \* To enable the pretty print
   FiniteSets,\* To understand IsFiniteSet()
   Integers   \* To resolve the operators, e.g. <, \div etc.
CONSTANTS
   I,    \* Identifiers (ID) for a node or a key
   A,    \* The initial ready nodes
   B,    \* For defining the base of ID in routing table
   M,    \* The maximal allowed exponents of 2,
         \*     which won't cause an overflow of natural number
   L     \* The parameter l in paper, max. length of each leafset
ASSUME IsFiniteSet(I)
ASSUME I \in SUBSET Nat
ASSUME A \in SUBSET I
ASSUME B < M /\ M % B = 0
ASSUME L >0
ASSUME Cardinality(A)>0
Base == 2^B
RingCap == 2^M       \* Identifier space
Dist(x, y) ==
   LET
       diff == x - y
       half == 2^(M-1)
   IN  IF diff < 0-half
        THEN diff + RingCap
        ELSE IF diff > half THEN diff - RingCap ELSE diff
AbsDist(x, i) ==
     LET hd == Dist(x, i) IN IF hd<0 THEN 0-hd ELSE hd
CwDist(x, i) ==
     IF Dist(x, i)  < 0
     THEN RingCap - AbsDist(x, i)
     ELSE AbsDist(x, i)
========================end of module Ring==========================
```

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 set, the other two components are the two leaf sets to either side of the node. The following operations access leaf sets.

Definition 2 (Operations on Leaf Sets)**.**

$$
\begin{aligned}
\mathit{GetLSetContent}(\mathit{ls}) &\triangleq \mathit{ls.left} \cup \mathit{ls.right} \cup \{\mathit{ls.node}\} \\
\mathit{LeftNeighbor}(\mathit{ls}) &\triangleq \begin{cases} \mathit{ls.node} & \mathit{if}\ \ \mathit{ls.left} = \{\} \\ \\ n \in \mathit{ls.left} : \forall p \in \mathit{ls.left} : & \\ \quad \mathit{CwDist}(p, \mathit{ls.node}) & \\ \quad \geq \mathit{CwDist}(n, \mathit{ls.node}) & \mathit{else} \end{cases} \\
\mathit{RightNeighbor}(\mathit{ls}) &\triangleq \begin{cases} \mathit{ls.node} & \mathit{if}\ \ \mathit{ls.right} = \{\} \\ \\ n \in \mathit{ls.right} : \forall q \in \mathit{ls.right} : & \\ \quad \mathit{CwDist}(\mathit{ls.node}, q) & \\ \quad \geq \mathit{CwDist}(\mathit{ls.node}, n) & \mathit{else} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{LeftCover}(\mathit{ls}) &\triangleq (\mathit{ls.node} + \mathit{CwDist}(\mathit{LeftNeighbor}(\mathit{ls}), \mathit{ls.node}) \div 2)\%2^M \\
\mathit{RightCover}(\mathit{ls}) &\triangleq (\mathit{RightNeighbor}(\mathit{ls}) + \\
&\quad \mathit{CwDist}(\mathit{ls.node}, \mathit{RightNeighbor}(\mathit{ls})) \div 2 + 1)\%2^M \\
\mathit{Covers}(\mathit{ls}, k) &\triangleq \mathit{CwDist}(\mathit{LeftCover}(\mathit{ls}), k) \\
&\quad \leq \mathit{CwDist}(\mathit{LeftCover}(\mathit{ls}), \mathit{RightCover}(\mathit{ls}))
\end{aligned}
$$

In these definitions, $\div$ and % stand for division and modulo on the natural numbers, respectively[2]. We also define the operation $\mathit{AddToLSet}(A, \mathit{ls})$ that updates the leaf set data structure with a set A of nodes. More precisely, both 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 A, according to the clockwise or counter-clockwise distance.

[2]Note that they are in fact only applied in the above definitions to natural numbers.

```
------------------------- MODULE LS ----------------------
EXTENDS Ring

LSet == {ls \in [node: I, left: SUBSET I, right: SUBSET I]:
         /\ ls.node \notin ls.left
         /\ ls.node \notin ls.right
         /\ Cardinality(ls.left) =< L
         /\ Cardinality(ls.right) =< L}

GetLSetContent(ls) == ls.left \cup ls.right \cup {ls.node}

RemoveFromLSet(delta, ls) ==
   [node |-> ls.node,
    left |-> ls.left \ delta,
   right |-> ls.right \ delta]


AddToLSet(delta, set) ==
  LET
    i == set.node
    left == ((set.left) \cup delta) \ {i}
    right == ((set.right) \cup delta) \ {i}
    newleft ==
      IF Cardinality(left) =< L
      THEN left
      ELSE CHOOSE subsetleft \in SUBSET left:
             /\ Cardinality(subsetleft) = L
             /\ \A out \in (left \ subsetleft),
                  in \in subsetleft:
                 CwDist(in,i) < CwDist(out,i)
    newright ==
      IF Cardinality(right) =< L
      THEN right
      ELSE CHOOSE subsetright \in SUBSET right:
               /\ Cardinality(subsetright) = L
               /\ \A out \in (right \ subsetright),
                     in \in subsetright:
                  CwDist(i,in) < CwDist(i,out)
  IN [node |-> i, left |-> newleft, right |-> newright]
```

```
LeftMost(ls)==(*find the leftmost node in lset wrt. I*)
    IF ls.left = {}
    THEN ls.node
    ELSE CHOOSE n \in ls.left:
           \A m \in ls.left: Dist(m, n) < 0 \/ n = m
RightMost(ls)==(*find the rightmost node in lset wrt. I*)
    IF  ls.right = {}
    THEN ls.node
    ELSE CHOOSE n \in ls.right:
     \A m \in ls.right: Dist(n, m) < 0 \/ n = m
LeftNeighbor(ls) ==
    IF ls.left = {}
    THEN ls.node
    ELSE CHOOSE n \in ls.left:
        \A p\in ls.left: CwDist(p,ls.node)>=CwDist(n,ls.node)
RightNeighbor(ls) ==
    IF ls.right = {}
    THEN ls.node
    ELSE CHOOSE n \in ls.right:
         \A p\in ls.right: CwDist(ls.node, p)>=CwDist(ls.node, n)
IsComplete(ls)==
    /\Cardinality(ls.left) = L
    /\Cardinality(ls.right) = L

InitLS(i) == AddToLSet(A, [node |-> i, left|-> {}, right|-> {}])
EmptyLS(i) == [node |-> i, left|-> {}, right|-> {}]
Overlaps(ls) == ls.left \cap ls.right # {}
Lenth(s)== Cardinality(s)
======================== end of module LS =========================

--------------------- MODULE RT (simlified)------------------------
EXTENDS LS
RTable == SUBSET I
InitRTable == {}\*Empty routing table
GetRTableContent(rt) == rt
AddToTable(newSet, rt, i) == rt \cup newSet
RemoveFromTable(set, rt, index) == rt \ set
======end of module RT
```

3.2 Dynamic Model

$$
\begin{array}{lcl}
\mathit{vars} & \triangleq & \langle \mathit{receivedMsgs}, \mathit{status}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{rtable} \rangle \\
\mathit{Init} & \triangleq & \land\ \mathit{receivedMsgs} = \{\} \\
 & & \land\ \mathit{status} = [i \in I \mapsto \textsc{if}\ i \in A\ \textsc{then}\ \text{“ready”}\ \textsc{else}\ \text{“dead”}] \\
 & & \land\ \mathit{lset} = [i \in I \mapsto \textsc{if}\ i \in A \\
 & & \qquad\qquad\qquad\qquad \textsc{then}\ \mathit{AddToLSet}(A, \mathit{EmptyLS}(i)) \\
 & & \qquad\qquad\qquad\qquad \textsc{else}\ \mathit{EmptyLS}(i)] \\
 & & \land\ \mathit{probing} = [i \in I \mapsto \{\}] \\
 & & \land\ \mathit{failed} = [i \in I \mapsto \{\}] \\
 & & \land\ \mathit{rtable} = [i \in I \mapsto \textsc{if}\ i \in A \\
 & & \qquad\qquad\qquad\qquad\quad \textsc{then}\ \mathit{AddToTable}(A, \mathit{InitRTable}, i) \\
 & & \qquad\qquad\qquad\qquad\quad \textsc{else}\ \mathit{AddToTable}(\{i\}, \mathit{InitRTable}, i)] \\
\mathit{Next} & \triangleq & \exists i, j \in I : \lor\ \mathit{Deliver}(i, j) \\
 & & \qquad\qquad\quad \lor\ \mathit{Join}(i, j) \\
 & & \qquad\qquad\quad \lor\ \mathit{JReply}(i, j) \\
 & & \qquad\qquad\quad \lor\ \mathit{Probe}(i, j) \\
 & & \qquad\qquad\quad \lor\ \mathit{PReply}(i, j) \\
 & & \qquad\qquad\quad \lor\ \ldots \\
\mathit{Spec} & \triangleq & \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}
\end{array}
$$

Figure 3.1: Overall Structure of the TLA$^+$ Specification of Pastry.

Figure 3.1 shows the high-level outline of the transition model specification in TLA$^+$. The overall system specification *Spec* is defined as $\mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}$, which is the standard form of TLA$^+$ system specifications. 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 formula *Next*. This form of system specification is sufficient for proving safety properties. If we were interested in proving liveness properties of our model, we should add fairness hypotheses asserting that certain actions eventually occur.

The variable *receivedMsgs* holds the set of messages in transit. Our model assumes that messages are never modified. However, message loss is implicitly covered because no action is ever required to execute. The other variables hold arrays that assign to every node $i \in I$ its status, leaf set, the set of nodes it is currently probing, the set of nodes it has determined to have dropped off the ring and routing table. The predicate *Init* is defined as a conjunction

$$
\begin{array}{l}
\mathit{Deliver}(i, k) \triangleq \\
\quad \land\ \mathit{status}[i] = \text{“ready”} \\
\quad \land\ \exists m \in \mathit{receivedMsgs} : \begin{array}[t]{l}
\land\ m.\mathit{mreq}.\mathit{type} = \text{“lookup”} \\
\land\ m.\mathit{destination} = i \\
\land\ m.\mathit{mreq}.\mathit{node} = k \\
\land\ \mathit{Covers}(\mathit{lset}[i], k) \\
\land\ \mathit{receivedMsgs}' = \mathit{receivedMsgs} \setminus \{m\}
\end{array} \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{status}, \mathit{rtable}, \mathit{lset}, \mathit{probing}, \mathit{failed} \rangle
\end{array}
$$

Figure 3.2: TLA$^+$ specification of action *Deliver*.

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[3]. for all pairs of identifiers $i, j \in I$. Each action is defined as a TLA$^+$ action formula, which 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 at the states before and after the action. As an example, Figure. 3.2 shows the definition of action $\mathit{Deliver}(i, k)$ in TLA$^+$. The action is executable if the node i is “ready”, if there exists an unhandled message of type “lookup” addressed to i, and if k, the ID of the requested key, falls within the coverage of node i (cf. Definition 2). Its effect is here simply defined as removing the message m from the network, because we are only interested in the execution of the action, not in the answer message that it generates. The other variables are unchanged (in TLA$^+$, UNCHANGED e is a shorthand for the formula $e' = e$).

```
------------------------- MODULE Msg -------------------------
EXTENDS RT
--------------------------------------------------------------
JReq == [type : {"JoinRequest"}, rtable: RTable, node: I]
JRpl == [type : {"JoinReply"}, rtable: RTable, lset: LSet]
Prb  == [type : {"LSProbe"}, node: I,
                             lset: LSet, failed: SUBSET I]
PRpl == [type : {"LSProbeReply"}, node: I,
                             lset: LSet, failed: SUBSET I]
Look == [type : {"Lookup"}, node: I]
NoLR == [type : {"NoLegalRoute"},  key: I]
```

[3]In this chapter, we only show the needed actions for demonstrating the counter example in section 3.4, other actions can be found later in section 4.1.

```
MReq == JReq \cup JRpl \cup Prb \cup PRpl
              \cup Look \cup NoLR
DMsg == [destination: I, mreq: MReq]
====================== end of module Msg====================
```

3.3 Validation By Model Checking

We used TLC [17], the TLA$^+$ model checker, to validate and debug our model. It is all too easy to introduce errors into a model that prevent the system from ever performing any useful transition, so we want to make sure that nodes can successfully perform *Deliver* actions or execute the join protocol described in Section 2. We used the model checker by asserting their impossibility, using the following formulas.

Property 1 (NeverDeliver and NeverJoin).

$$\begin{array}{lcl} \mathit{NeverDeliver} & \triangleq & \forall i, j \in I : \Box[\neg \mathit{Deliver}(i, j)]_{\mathit{vars}} \\ \mathit{NeverJoin} & \triangleq & \forall j \in I \setminus A : \Box(\mathit{status}[j] \neq \text{``}\mathit{ready}\text{''}) \end{array}$$

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”. Running the model checker on our model, it quickly produced counter-examples to these claims, which we examined to ensure that the runs look as expected. (Section 4.3 summarizes the results for the model checking runs that we performed.)

We validated the model by checking several similar properties. For example, we defined formulas *ConcurrentJoin* and *CcJoinDeliver* whose violation yielded counter-examples that show 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.

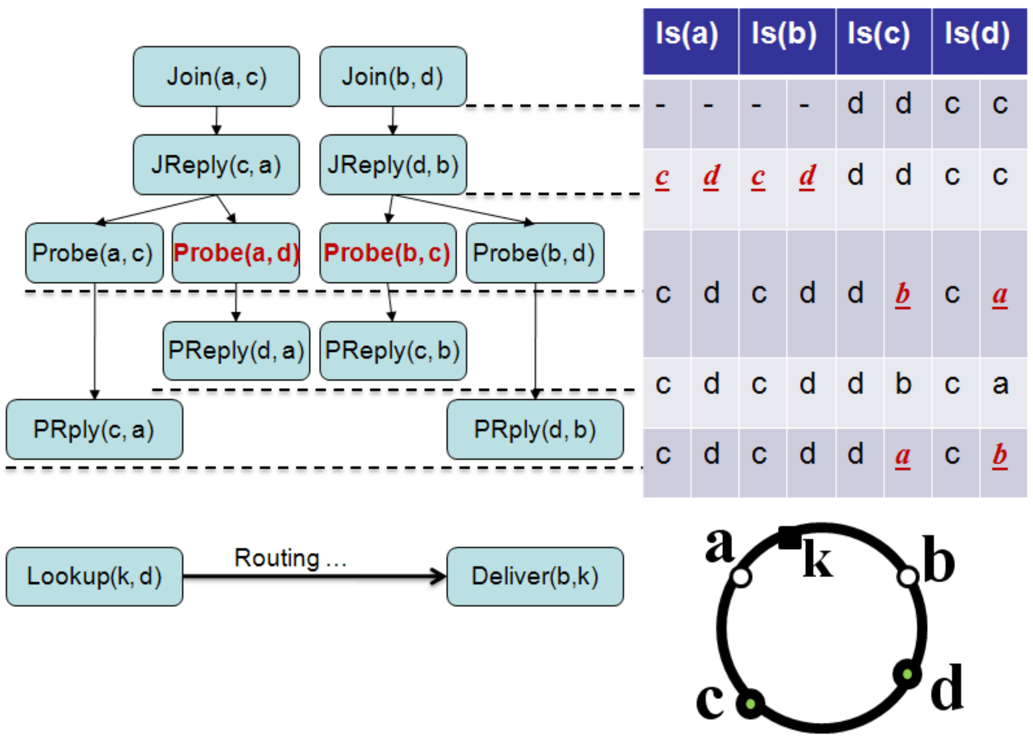
3.4 Correct Key Delivery

As the main correctness property of Pastry, we want to show that at any time there can be only one node responsible for any key. This is formally expressed as follows.

Property 2 (Correct Key Delivery).

$$
\begin{array}{l}
\mathit{CorrectDelivery} \triangleq \forall i, k \in I : \\
\quad \text{ENABLED } \mathit{Deliver}(i, k) \\
\quad \Rightarrow \land\ \forall n \in I : \mathit{status}[n] = \text{“}\mathit{ready}\text{”} \Rightarrow \mathit{AbsDist}(i, k) \leq \mathit{AbsDist}(n, k) \\
\qquad \land\ \forall j \in I \setminus \{i\} : \neg \text{ENABLED } \mathit{Deliver}(j, k)
\end{array}
$$

For an action formula A, the state formula ENABLED 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 is true for the pair (s, t), that is, when A can execute at state s. Thus, *CorrectDelivery* asserts that whenever node i can execute the *Deliver* action for key k then (a) node i has minimal absolute distance from k among all the “ready” nodes and (b) i is the only node that may execute *Deliver* for key k.[4]



Is(a)		Is(b)		Is(c)		Is(d)	
-	-	-	-	d	d	c	c
c	d	c	d	d	d	c	c
c	d	c	d	d	b	c	a
c	d	c	d	d	b	c	a
c	d	c	d	d	a	c	b

Figure 3.3: Counter-example leading to a violation of *CorrectDelivery*.

[4]Observe that there can be two nodes with minimal distance from k, to either side of the key. The asymmetry in the definition of *LeftCover* and *RightCover* is designed to break the tie and ensure that only one node is allowed to deliver.

When we attempted to verify Property 2, the model checker produced a counter-example, which we illustrate in Figure 3.3. The run starts in a state with just two "ready" nodes c and d that contain each other in their respective leaf sets (the actual size of the leaf sets being 1). Two nodes a and b concurrently join between nodes c and d. According to their location on the ring, a's join request is handled by node c, and b's request by d. Both nodes 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 the leaf sets. Now, suppose that node d is the first to handle a's probe message, and that node c first handles b's probe. Learning that a new node has joined, which is closer than the previous entry in the respective leaf set, c and d update their leaf sets with b and a, respectively (cf. Figure 3.3), and send these updated leaf sets to b and a. Based on the reply from d, node a will not update its leaf set 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 a's probe: it learns of the existence of a new node to its right closer to the one currently in its leaf set (b) and updates its leaf set accordingly, then replies to a. However, node a still does not learn about node b from this reply and maintains its leaf sets containing c and d. Symmetrically, node d updates its leaf set to contain b instead of a, but b does not learn about the presence of a. At the end, the leaf sets of the old nodes c and d are correct, but a and b do not know about each other and have incorrect leaf set entries.

Finally, a lookup message arrives for key k, 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, which b believes to be its left-hand neighbor), and delivers the key.

The counter-example shows that our model of the join protocol may lead to inconsistent views of "ready" nodes about their neighborhoods on the ring, and is therefore insufficient. Indeed, after the initial publication of Pastry, Haeberlen et al. [8] presented a refined description of Pastry's join protocol, without providing an explicit motivation. We believe that the above counter example explains the refinement of [8], which we model and analyze in the sequel.

4 Refining the Join Protocol

In contrast to the join protocol described in Section 2, the refined join protocol requires an explicit transfer of coverage from the "ready" neighbor nodes before a joining node can become "ready" and answer lookup requests. In the case of the counter-example shown in Figure 3.3, node *a* would request grants from the nodes *c* and *d*, which it believes to be its neighbors. Node *d* would refuse this request and instead inform node *a* of the presence of node *b*, enabling it to rebuild its leaf sets. Similarly, node *b* would learn about the presence of node *a*. Finally, the two nodes grant each other a lease for the nodes they cover. We now describe the extended protocol as we have understood and modeled it, and our further verification efforts. In fact, our formal model is also inspired by the implementation in FreePastry [13], where nodes periodically exchange their leaf sets to spread information about nodes dropping off and arriving.

4.1 Lease Granting Protocol

Figure 4.1 depicts the extension to the join protocol as described in Section 2 (cf. Figure 2.1). After node *i* has built complete leaf sets, it reaches status "ok". It sends 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 a node that it considers to be its neighbor grants the lease, otherwise it returns its own leaf sets to the requesting node. The receiving node will update its own leaf sets accordingly and request a lease from the new neighbor(s). Only when both neighbors grant the lease will node *i* become "ready".

Moreover, any node that is "ok" or "ready" may non-deterministically re-run the lease granting protocol at any time. In the actual implementation, this happens periodically, as well as when a node suspects its neighbor to have left the ring.

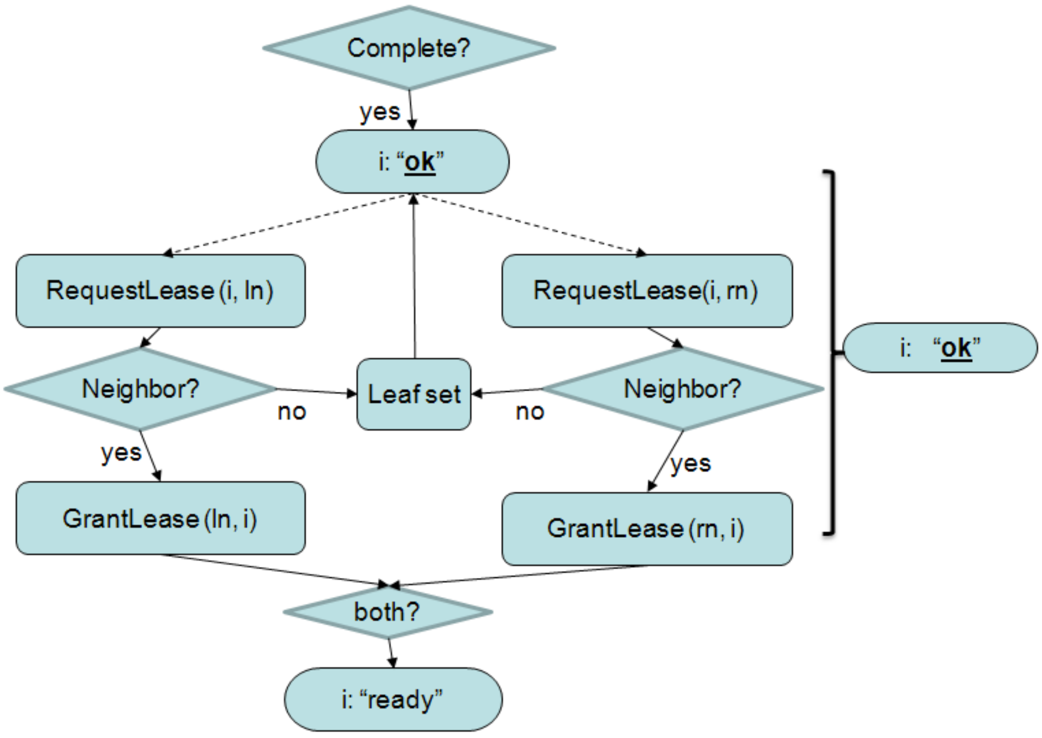


Figure 4.1: Extending the Join Protocol by Lease Granting.

We amended our TLA$^+$ model to reflect this extended join protocol and reran TLC on the extended model. Whereas the results for the properties used to validate the model (cf. Section 3.3) were unchanged, the model checker no longer produced a counter-example to Property 2. However, we were unable to complete the model checking run and killed TLC after it had been running for more than a month.

The extended TLA$^+$ model has two more variables (cf. Figure 4.2), *lease* stores the nodes from which it has already got the leases and *grant* stores the nodes to which it has granted its leases. The message type has been extended with *LReq* for sending the lease request and *BLS* for broadcasting the leaf set and at the same time granting the lease if applicable (cf. TLA$^+$ module Msg).

```
------------------------- MODULE Msg ------------------------
......
LReq == [type : {"LeaseRequest"}, node: I]
BLS  == [type : {"BroadcastLSet"}, lset: LSet, grant: BOOLEAN]
MReq == ... \cup LReq \cup BLS
================ end of extended module of Msg ===============
```

$$
\begin{array}{lll}
\mathit{vars} & \triangleq & \langle \mathit{receivedMsgs}, \mathit{status}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{rtable}, \mathbf{lease}, \mathbf{grant} \rangle \\
\mathit{Init} & \triangleq & \land \vdots \\
 & & \land\ \mathbf{lease} = [\mathbf{i} \in \mathbf{I} \mapsto \{\}] \\
 & & \land\ \mathbf{grant} = [\mathbf{i} \in \mathbf{I} \mapsto \{\}] \\
\mathit{Next} & \triangleq & \exists i, j \in I : \lor \vdots \\
 & & \qquad\qquad \lor\ \mathbf{RequestLease(i)} \lor \mathbf{ReceiveLReq(i)} \lor \mathbf{ReceiveBLS(i)} \\
 & & \qquad\qquad \lor\ \mathbf{LeaseExpired(i,j)} \lor \mathbf{DeclareDead(i,j)} \\
\mathit{Spec} & \triangleq & \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}
\end{array}
$$

Figure 4.2: Extension of overall structure shown in Figure 3.1.

Here we show the formal TLA$^+$ model of Pastry routing protocol. In the module *InitialStates* we define the type invariants and initialization of the variables.

In the module *Actions* we define the actions a node will execute. An action consist of preconditions and effects. The primed variables stand for the changes of the variables as effects of the actions. In the following, we explain in particular those actions in detail that are crucial for the properties studied in the paper. The prefix *Receive* of many action names denotes that those actions are reactive on the message it received, which has the same name as the postfix of the action name. For example, one of the precondition of action *ReceiveLSProbe* is that a node i received a message of type *LSProbe* and its destination is i.

```
------------------------ MODULE InitialStates -----------------
EXTENDS Msg
VARIABLE
   receivedMsgs, status, lease, grant,
   rtable, lset, probing, failed
Init ==
  /\ receivedMsgs = {}
  /\ status = [i \in I |-> IF i \in A THEN "ready" ELSE "dead"]
  /\ rtable = [i \in I |-> IF i \in A
                           THEN AddToTable(A, InitRTable, i)
                           ELSE AddToTable({i}, InitRTable, i)]
  /\ lset   = [i \in I |-> IF i \in A
                           THEN AddToLSet(A, EmptyLS(i))
                           ELSE EmptyLS(i)]
  /\ probing= [i \in I |-> {}]
  /\ failed = [i \in I |-> {}]
  /\ lease  = [i \in I |-> IF i \in A THEN A  ELSE {}]
  /\ grant  = [i \in I |-> IF i \in A THEN A ELSE {}]
TypeInvariant ==
  /\ receivedMsgs \in SUBSET DMsg
  /\ status \in [I -> {"ready", "ok", "wait", "dead"}]
  /\ lease  \in [I -> SUBSET I] \* from which nodes does a node got lease
  /\ grant  \in [I -> SUBSET I] \* to which nodes does a node grant the lease
  /\ rtable \in [I -> RTable]
  /\ lset   \in [I -> LSet] /\ \A i \in I: lset[i].node = i
  /\ probing\in [I -> SUBSET I]
  /\ failed \in [I -> SUBSET I]
=====================end of module IntialStates=========================
--------------------- MODULE Actions ---------------------
EXTENDS InitialStates

Lookup(i, k) ==
  /\ ~\E m \in receivedMsgs:
     /\ m.destination = i
     /\ m.mreq.type   = "Lookup"
     /\ m.mreq.node   = k
  /\ receivedMsgs'    = receivedMsgs \cup
           {[destination |-> i,
             mreq |-> [type |-> "Lookup",  node |-> k]]}
  /\ UNCHANGED <<status, rtable, lset, probing,
                     failed, lease, grant>>
Join(i, seed) ==
  /\ i # seed
  /\ ~\E m \in receivedMsgs:
     /\ m.mreq.type = "JoinRequest"
     /\ m.mreq.node = i
  /\ status[i]    = "dead"
  /\ status[seed] ="ready"
  /\ lset[i]      = EmptyLS(i)
  /\ receivedMsgs'= receivedMsgs \cup
                   {[destination |-> seed,
                     mreq |-> [type |-> "JoinRequest",
                             rtable |-> InitRTable,
                               node |-> i] ]}
  /\ status'      = [status EXCEPT ![i]="wait"]
  /\ UNCHANGED <<rtable, lset, probing,
                      failed, lease, grant>>

Deliver(i, k) ==
  /\ status[i] = "ready"
  /\ \E m \in receivedMsgs:
     /\ m.mreq.type   = "Lookup"
     /\ m.destination = i
     /\ m.mreq.node   = k
     /\ Covers(lset[i], k)
     /\ receivedMsgs' = (receivedMsgs \{m\})
  /\ UNCHANGED <<status, rtable, lset, probing,
                     failed, lease, grant>>
```

We define the following function *FindNext* $\in I \times I \mapsto I$ to find the next hop for routing. We modeled the liveness checking of a node c (via Ping) by reading the value of *status*$[c]$ if it is "dead" or not. A node could leave the network or the communication of its channel could be blocked, by these situation, its status is "dead". We omitted the routing table look up, because we don't use the model to prove any property of the routing table.

```
FindNext(k, i) ==
  LET
     lsCan ==  {c \in GetLSetContent(lset[i]) \ failed[i]:
                     status[c] # "dead"}
     canrelax ==
           {can \in (GetLSetContent(lset[i])
                    \cup GetRTableContent(rtable[i])) \ failed[i]:
                         /\ AbsDist(k, can) < AbsDist(k, i)
                         /\ status[can] # "dead"}
  IN IF /\ (\/ Overlaps(lset[i])
            \/ CwDist(LeftMost(lset[i]), i)
              =< CwDist(LeftMost(lset[i]), RightMost(lset[i])))
        /\ ~ lsCan = {}
     THEN CHOOSE n \in lsCan: \A m \in lsCan:
               AbsDist(n, k) =< AbsDist(m, k)
     ELSE  IF ~canrelax = {}
           THEN CHOOSE can \in canrelax: \A m \in canrelax:
                      AbsDist(can, k) =< AbsDist(m, k)
           ELSE i
RouteLookup(i, k) ==
  /\ status[i] = "ready"
  /\ \E m \in receivedMsgs:
     /\ m.destination = i
     /\ m.mreq.type   = "Lookup"
     /\ m.mreq.node   = k
     /\ ~Covers(lset[i], k)
     /\ LET nh ==  FindNext(k, i)
        IN  receivedMsgs'= (receivedMsgs \{m\}) \cup
              IF nh # i
              THEN {[destination |-> nh, mreq |-> m.mreq]}
              ELSE {[destination |-> i,  mreq |->
                        [type|->"NoLegalRoute", key|->k]]}
  /\ UNCHANGED <<status, rtable, lset, probing,
                          failed, lease, grant>>
```

```
RouteJoinRequest(i, k) ==
  /\ status[i] = "ready"
  /\ \E m \in receivedMsgs:
     /\ m.destination = i
     /\ m.mreq.type = "JoinRequest"
     /\ m.mreq.node = k
     /\ ~k \in GetLSetContent(lset[i])
     \* to avoid the case that a node left and
     \* rejoin before its neighbor deleted it from their leaf set
     /\ ~Covers(lset[i], k)
     /\ LET nh ==  FindNext(k, i)
        IN
        receivedMsgs'= (receivedMsgs \{m\}) \cup
          IF nh # i
          THEN {[destination |-> nh, mreq |->
                  [type |-> "JoinRequest",
                 rtable |-> AddToTable(GetRTableContent(rtable[i]),
                                                m.mreq.rtable, i),
                   node |-> k]]}
          ELSE {[destination |-> i, mreq |->
                  [type |-> "NoLegalRoute", key |-> k]]}
  /\ UNCHANGED <<status, rtable, lset, probing, failed, lease, grant>>

ReceiveJoinRequest(i) ==
  /\ status[i] = "ready"
  /\ \E m \in receivedMsgs:
   /\ m.mreq.node # i
   /\ m.destination = i
   /\ m.mreq.type = "JoinRequest"
   /\ ~m.mreq.node \in GetLSetContent(lset[i])
   \* to avoid the case that a node left and
   \*rejoin before its neighbor deleted it from their leaf set
   /\ CwDist(LeftCover(lset[i]), m.mreq.node)
       =< CwDist(LeftCover(lset[i]), RightCover(lset[i]))
   /\ LET newmjr ==      [ type |-> "JoinReply",
                        rtable |-> m.mreq.rtable,
                          lset |-> lset[i]]
          newmj == [destination |-> m.mreq.node,
                          mreq |-> newmjr]
      IN receivedMsgs'= (receivedMsgs \{m\}) \cup {newmj}
  /\ UNCHANGED <<status, rtable, lset, probing, failed, lease, grant>>
```

The precondition of action *ReceiveJoinReply*(i) requires that the node not to be "dead", because receiving a JoinReply message as a "dead" node has obviously no effect. It requires probing to be empty because a received JoinReply is a precondition of probing and according to the nature of the protocol, a node could receive at most only one JoinReply at each time slot.

The received *JoinReply* message is removed, the node updates its leaf set with the received leaf set and send *LSProbe* message to the new members in its new leaf set.

```
Probe(i, ls, f, toprob) ==
  {[destination |-> j,
          mreq |-> [type |-> "LSProbe",
                    node |-> i,
                    lset |-> ls,
                  failed |-> f
                   ]
  ]: j\in toprob}


ReceiveJoinReply(i) ==
/\ status[i]  # "dead"
/\ probing[i] = {}
/\ lset[i]    = EmptyLS(i)
/\ \E m \in receivedMsgs:
   /\ m.destination = i
   /\ m.mreq.type = "JoinReply"
   /\ LET
        newrtable== AddToTable(GetLSetContent(m.mreq.lset)
                              \cup GetRTableContent(m.mreq.rtable),
                             rtable[i], i)
         newlset == AddToLSet(GetLSetContent(m.mreq.lset), lset[i])
          toprob == GetLSetContent(newlset) \ {i}
         probmsgs== Probe(i, newlset, {}, toprob)
       IN
      /\ rtable' = [rtable EXCEPT ![i] = newrtable]
      /\ lset'   = [lset EXCEPT ![i] = newlset]
      /\ probing'= [probing EXCEPT ![i] = toprob]
      /\ failed' = [failed EXCEPT ![i] = {}]
      /\ receivedMsgs'= (receivedMsgs \ {m}) \cup probmsgs
/\ UNCHANGED <<status, lease, grant>>
```

The precondition of the actions *ReceiveLSProbe* or *ReceiveLSProbeReply* requires the leaf set of the node not to be empty, if the node has the status of "wait", which means that it has sent the *JoinRequest* some time in the past and has not yet completed its leaf set. This precondition ensures that this action can be executed only after a node has already executed the action *ReceiveJoinReply*.

When receiving *LSProbe* or *LSProbeReply*, the node will add the sender ID into its leaf set. If the received message m contains new candidates to be member of i's leaf set, then i will send *LSProbe* to them.

By receiving an *LSProbe*, the node will send the *LSProbeReply* back with its *updated* leaf set. Here the $prb1$ stands for the leaf set members from which the sender j believes they are failed. The node i will then check them itself. The $prb2$ stands for potential new leaf set members after adding received leaf set and not yet probed.

```
ReceiveLSProbe(i) ==
/\ status[i] # "dead"
/\ status[i] = "wait" => lset[i] # EmptyLS(i)
/\ \E m \in receivedMsgs:
  /\ m.destination = i
  /\ m.mreq.type = "LSProbe"
  /\ m.mreq.node = m.mreq.lset.node
  /\ m.mreq.node # i
  /\ LET j == m.mreq.node
        fi == failed[i] \ {j}
       ls1 == IF j \in GetLSetContent(lset[i])
              THEN lset[i]
              ELSE AddToLSet({j}, lset[i])
     lprim == GetLSetContent(
                   AddToLSet(
                   (GetLSetContent(m.mreq.lset) \ fi), ls1))
      prb1 == GetLSetContent(ls1) \cap m.mreq.failed
      prb2 == lprim \ GetLSetContent(ls1)
      prb  == (prb1 \cup prb2) \ (probing[i] \cup fi)
      newm == [type |-> "LSProbeReply",
                  node |-> i,
                  lset |-> ls1,
                  failed |-> fi]
    IN
         /\failed' = [failed EXCEPT ![i]=fi]
         /\rtable' = [rtable EXCEPT ![i] =
                              AddToTable({j}, @, i)]
         /\lset'   = [lset EXCEPT ![i] = ls1]
         /\probing'= [probing EXCEPT ![i] = @ \cup prb]
         /\receivedMsgs' = (receivedMsgs \ {m})
                           \cup { [destination |-> j,
                                         mreq |-> newm]}
                           \cup Probe(i, ls1, fi, prb)
/\ UNCHANGED <<lease, status, grant>>
```

By receiving *LSProbeReply*, only the message from the nodes it is currently probing are handled, to avoid garbage message making any confusion. Here the *prb*3 stands for the overall nodes to probe or being probed. The status of the node will become "ok", if (i) the node is still "wait"; (ii) *prb*3 is empty, meaning for all nodes it has probed, it has got a reply and there is no more node need to probe; (iii) each of the leaf set has the length of L or the left and right leaf set have common members.

```
ReceiveLSPrRpl(i) ==
/\ status[i] # "dead"
/\ status[i] = "wait" => lset[i] # EmptyLS(i)
/\ \E m \in receivedMsgs:
   /\ m.destination = i
   /\ m.mreq.type = "LSProbeReply"
   /\ m.mreq.node \in probing[i]
   /\ m.mreq.node # i
   /\LET j== m.mreq.node
        Ls == m.mreq.lset
        fi == failed[i] \ {j}
       ls1 == IF j \in GetLSetContent(lset[i])
              THEN lset[i]
              ELSE AddToLSet({j}, lset[i])
     lprim == GetLSetContent(
                 AddToLSet((GetLSetContent(Ls) \ fi), ls1))
      prb1 == (GetLSetContent(ls1) \cap m.mreq.failed )
                   \ (probing[i] \cup fi)
      prb2 == lprim \ (GetLSetContent(ls1) \cup probing[i]
                   \cup fi \cup prb1)
      prb3 == ((probing[i] \cup prb1 \cup prb2)
                   \ failed[i]) \ {j}
```

```
shouldBeOK== /\ status[i] = "wait"
             /\ prb3={}
             /\ Overlaps(ls1)\/IsComplete(ls1)
   IN
     /\ rtable' = [rtable EXCEPT ![i] =
                         AddToTable({j}, @, i)]
     /\ lset'   = [lset EXCEPT ![i] = ls1]
     /\ failed' = [failed EXCEPT ![i] =
                   IF prb3={} /\ IsComplete(ls1) THEN {} ELSE fi]
     /\ probing'= [probing EXCEPT ![i] = prb3]
     /\ status' = [status EXCEPT ![i] =
                   IF shouldBeOK THEN "ok" ELSE @]
     /\ receivedMsgs' = (receivedMsgs \ {m})
                        \cup Probe(i, ls1, fi, prb1)
                        \cup Probe(i, ls1, fi, prb2)
 /\ UNCHANGED <<lease, grant>>
```

The following actions are involved to handle the case, when a node has left and how the other nodes react on it. For the moment we assume only "ready" nodes can leave, and we assume that no nodes will leave the network if there is less than $L + 1$ nodes working.

```
NodeDied(i) ==
  /\ status[i] = "ready"
  /\ Cardinality({k \in I: status[k] = "ready"})> L + 1
  /\ status' = [status EXCEPT ![i] = "dead"]
  /\ rtable' = [rtable EXCEPT ![i] = AddToTable({i}, InitRTable, i)]
  /\ lset'   = [lset EXCEPT ![i] = EmptyLS(i)]
  /\ probing'= [probing EXCEPT ![i] = {}]
  /\ failed' = [failed EXCEPT ![i] = {}]
  /\ lease'  = [lease EXCEPT ![i]= {}]
  /\ grant'  = [grant EXCEPT ![i]= {}]
  /\ UNCHANGED <<receivedMsgs>>
```

When a node dies, the nearby nodes might non-deterministically suspect it to be faulty and hence probe it. It doesn't make sense for a waiting node to suspect any dead node.

```
SuspectFaulty(i, j) ==
  /\ status[i] \in {"ready", "ok"}
  /\ status[j]= "dead"
  /\ i # j
  /\ j \in GetLSetContent(lset[i])
  /\ j \notin probing[i]
  /\ j \notin failed[i]
  /\ receivedMsgs'= receivedMsgs \cup Probe(i, lset[i], failed[i], {j})
  /\ probing' = [probing EXCEPT ![i]= @\cup {j}]
  /\ UNCHANGED <<rtable, lset, failed, lease, grant, status>>
```

We simulate the timeout as followed. Here we delete all the garbage probe messages from i to j, since we are sure no one will get a reply any more. Notice that only if someone from the current leaf set of i has left the network, i will change its status. One can only handle the non-responding node, after its lease is expired.

```
ProbeTimeOut(i, j)==
/\ status[i]# "dead"
/\ \/ status[j] = "dead"
   \/ /\ status[j] = "wait"
      /\ lset[j]   = EmptyLS(j)
/\ j \in probing[i]
/\ failed'  = [failed EXCEPT ![i]  = @\cup {j}]
/\ probing' = [probing EXCEPT ![i] = @ \ {j}]
/\ lset'    = [lset EXCEPT ![i] =
               IF ~(j \in grant[i]) /\ j \in GetLSetContent(@)
               THEN RemoveFromLSet({j},@)
               ELSE @]
/\ receivedMsgs' = receivedMsgs \
               IF \E m \in receivedMsgs: m.mreq.type = "LSProbe"
                    /\ m.mreq.node = i
                    /\ m.destination = j
               THEN  {m \in receivedMsgs: m.mreq.type = "LSProbe"
                         /\ m.mreq.node = i
                         /\ m.destination = j}
               ELSE {}
/\ status' = [status EXCEPT ![i] =
               IF /\ ~(j \in grant[i])
                  /\ j \in GetLSetContent(lset[i])
               THEN "wait" ELSE @]
/\ lease'  = [lease EXCEPT ![i] = @ \ {j}]
/\ UNCHANGED <<rtable, grant>>
```

```
RequestLease(i) ==
    /\ status[i] = "ok"
    /\ lset[i] # EmptyLS(i)
    /\ LET ln == LeftNeighbor(lset[i])
           rn == RightNeighbor(lset[i])
       IN receivedMsgs' = (receivedMsgs
           \cup IF ~ (ln \in lease[i])
                THEN {[destination |-> ln,
                      mreq |-> [type |-> "LeaseRequest",
                                node |-> i]]}
                ELSE {})
           \cup IF ~ (rn \in lease[i])
                THEN {[destination |-> rn,
                      mreq |-> [type |-> "LeaseRequest",
                                node |-> i]]}
                ELSE {}
    /\ UNCHANGED <<status, lset, rtable, probing,
                   failed, lease, grant>>
ReceiveLReq(i) ==
  /\ status[i] \in {"ready", "ok"}
  /\ lset[i] # EmptyLS(i)
  /\ \E m \in receivedMsgs:
   /\ m.destination = i
   /\ m.mreq.type = "LeaseRequest"
   /\ grant' = [grant EXCEPT ![i] =
                  IF m.mreq.node
                     \in {LeftNeighbor(lset[i]),
                          RightNeighbor(lset[i])}
                  THEN @ \cup {m.mreq.node}
                  ELSE @]
    /\ receivedMsgs' = (receivedMsgs \ {m})
                  \cup {[destination |-> m.mreq.node,
                     mreq |-> [type |-> "BroadcastLSet",
                               lset |-> lset[i],
                               grant|->
                                    m.mreq.node \in
                                   {LeftNeighbor(lset[i]),
                                    RightNeighbor(lset[i])}]]}
  /\ UNCHANGED <<status, rtable, lset, probing, failed, lease>>
```

```
ReceiveBLS(i) ==
  /\ status[i] \in {"ready", "ok"}
  /\ lset[i] # EmptyLS(i)
  /\ IsComplete(lset[i])
  /\ \E m \in receivedMsgs:
   /\ m.destination = i
   /\ m.mreq.type = "BroadcastLSet"
   /\ m.mreq.lset.node # i
   /\ LET
          ls == AddToLSet(
                 GetLSetContent(m.mreq.lset)
                     \failed[i] , lset[i])
          ln == LeftNeighbor(ls)
          rn == RightNeighbor(ls)
    newlease == [lease EXCEPT ![i] =
                 IF /\ m.mreq.lset.node \in {ln, rn}
                    /\ m.mreq.grant = TRUE
                 THEN @ \cup {m.mreq.lset.node}
                 ELSE @]
shouldBeReady== /\ ln \in newlease[i]
                /\ rn \in newlease[i]
                /\ ln = LeftNeighbor(lset[i])
                /\ rn = RightNeighbor(lset[i])
      IN /\ lset' = [lset EXCEPT ![i] = ls]
         /\ lease' = newlease
         /\ status' = [status EXCEPT ![i]=
              IF shouldBeReady THEN "ready" ELSE @ ]
         /\ receivedMsgs' = (receivedMsgs \ {m})
  /\ UNCHANGED <<rtable, probing, failed, grant>>
LeaseExpired(i, j) ==
  /\ status[i] \in {"ready", "ok"}
  /\ i # j
  /\ j \in lease[i]
  /\ lease'  = [lease EXCEPT ![i] = @ \{j\}]
  /\ status' = [status EXCEPT ![i] =
        IF /\ j \in {LeftNeighbor(lset[i]),
                     RightNeighbor(lset[i])}
           /\ @ = "ready"
        THEN "ok" ELSE @]
  /\ UNCHANGED <<receivedMsgs, rtable,
              lset, probing, failed, grant>>
```

```
GrantExpired(i, j) ==
  /\ i # j
  /\ status[i] \in {"ready", "ok"}
  /\ j \in grant[i] /\ ~ (i \in lease[j])
  /\ grant'= [grant EXCEPT ![i] = @ \ {j}]
  /\ UNCHANGED <<receivedMsgs, status,
            lease, lset, probing, failed, rtable>>
DeclareDead(i,j) ==
  /\ status[i] \in {"ready", "ok"}
  /\ j \in failed[i] \* After i has put j in failed for a while.
  /\ j \in GetLSetContent(lset[i])
  /\ j = LeftNeighbor(lset[i]) => ~ j \in grant[i]
  /\ j = RightNeighbor(lset[i])=> ~ j \in grant[i]
  /\ lset'  = [lset EXCEPT ![i] = RemoveFromLSet({j},@)]
  /\ status'= [status EXCEPT ![i] = "wait"]
  /\ UNCHANGED <<receivedMsgs, rtable,
               lease, probing, failed, grant>>
```

Further, we modeled the following actions to repair the leaf set when half of the leaf set members of a node drop off the network.

```
ResignNode(i) ==
  /\ status[i] = "wait"
  /\ probing[i] = {}
  /\ Lenth(lset[i].left) = 0  \/ Lenth(lset[i].right) = 0
  /\ ~\E m \in receivedMsgs:
      m.mreq.type = "JoinReply" /\ m.destination = i
  /\ \E n \in I: n#i /\ status[n] = "ready"
    /\ rtable' = [rtable  EXCEPT ![i] = AddToTable({i},InitRTable,i)]
    /\ lset'   = [lset    EXCEPT ![i] = EmptyLS(i)]
    /\ probing'= [probing EXCEPT ![i] = {}]
    /\ lease'  = [lease   EXCEPT ![i] = {}]
    /\ grant'  = [grant   EXCEPT ![i] = {}]
    /\ failed' = [failed  EXCEPT ![i] = {}]
    /\ receivedMsgs' = receivedMsgs \cup
                     {[destination |-> n,
                       mreq |-> [type |-> "JoinRequest",
                               rtable |-> InitRTable,
                                 node |-> i]]}
```

```
Recover(i)==
  /\ status[i] = "wait"
  /\ probing[i] = {}
  /\ Lenth(lset[i].left) = 0   \/ Lenth(lset[i].right) = 0
  /\ ~\E n \in I:  status[n] \in {"ready", "ok"}
  /\ ~(Lenth(lset[i].left) = 0 /\ Lenth(lset[i].right) = 0)
  /\ LET
      pe == IF   Lenth(lset[i].left) = 0
            THEN RightMost(lset[i])
            ELSE LeftMost(lset[i])
   IN
    /\ probing' = [probing EXCEPT ![i] =
                           probing[i] \cup {pe}]
    /\ receivedMsgs' = receivedMsgs
                    \cup Probe(i, lset[i], failed[i], {pe})
  /\ UNCHANGED <<lset, rtable, failed, lease, status, grant>>

LSRepair(i) ==
  /\ status[i] = "wait"
  /\ probing[i] = {}
  /\ ~IsComplete(lset[i])
  /\ Lenth(lset[i].left) # 0 /\ Lenth(lset[i].right) # 0
  /\ LET
   lm == {LeftMost(lset[i])}
   rm == {RightMost(lset[i])}
   newprob == IF /\ Lenth(lset[i].left)  < L
                 /\ Lenth(lset[i].right) < L
              THEN lm \cup rm
              ELSE IF /\ Lenth(lset[i].left)  < L
                      /\ Lenth(lset[i].right) = L
                   THEN lm ELSE rm
     IN
    /\ probing' = [probing EXCEPT ![i]=probing[i] \cup newprob]
    /\ receivedMsgs' = receivedMsgs
                \cup Probe(i, lset[i], failed[i], newprob)
  /\ UNCHANGED <<lset, rtable, failed, lease, status, grant>>
====================end of module action=========================
```

Examples	Time	Depth	# states	Counter Example
NeverDeliver	1”	5	101	yes
NeverJoin	1”	9	19	yes
ConcurrentJoin	3’53”	21	212719	yes
CcJoinDeliver	23’16”	23	1141123	yes
Symmetry	17”	17	19828	yes
Neighbor	5’35”	16	278904	yes
NeighborProp	> 1 month	31	1331364126	no
CorrectDeliver	> 1 month	21	1952882411	no

Table 4.1: TLC result with four nodes, leaf set length $l = 1$

4.2 Symmetry of Leaf Sets

Based on the counter-example shown in Section 3.4, one may be tempted to assert that leaf set membership of nodes should be symmetrical in the sense that for any two “ready” nodes i, j it holds that i appears in the leaf sets of j if and only if j appears in the leaf sets of i.

Property 3 (Symmetry of leaf set membership).

$$\begin{array}{l} \mathit{Symmetry} \triangleq \\ \quad \forall i, j \in I : \mathit{status}[i] = \text{“ready”} \wedge \mathit{status}[j] = \text{“ready”} \\ \qquad\qquad \Rightarrow (i \in \mathit{GetLSetContent}(\mathit{lset}[j]) \Leftrightarrow j \in \mathit{GetLSetContent}(\mathit{lset}[i])) \end{array}$$

However, the above property is violated during the execution of the join protocol and TLC yields the following counter-example: a node k joins between i and j. It finishes its communication with i getting its coverage set from i, but its communication with j is not yet finished. Hence, i and j are “ready” whereas k is not. Furthermore, i may have removed j from its leaf set, so the symmetry is broken.

4.3 Validation

Table 4.1 summarizes the model checking experimentswe have described so far, over the extended model. TLC was run with two worker threads (on two CPUs) on a 32 Bit Linux machine with Xeon(R) X5460 CPUs running at 3.16GHz with about 4 GB of memory per CPU. For each run, we report the running time, the number of states generated until TLC found a counter-example (or, in the case of Property 2, until we killed the process), and the largest depth of these states. Since the verification of Property 2 did not produce a counter-example, we ran the model checker in breadth-first search mode. We can therefore assert that if the model contains a counter-example to this property, it must be of depth at least 21.

All properties except *Neighbor* and *NeighborProp* were introduced in previous sections. The property *Neighbor* is inspired by the counter-example described in Section 3.4. It is actually the *NeighborClosest* property relaxed to "ok" and "ready" nodes. It asserts that whenever i, j are nodes that are "ok" or "ready", then the left and right neighbors of node i according to its leaf set contents must be at least as close to i than is node j. This property does not hold, as the counter-example of Section 3.4 shows, but it does if node i is in fact "ready", which corresponds the *NeighborClosest* property. The *NeighborProp* property is the conjunction *HalfNeighbor* $\wedge$ *NeighborClosest*, see the next section (or the TLA$^+$ formal model attached below).

```
--------------------------MODULE MCMSPastry---------------------
EXTENDS Actions
vars == <<receivedMsgs, status, rtable,
             lset, probing, failed, lease, grant>>
MCI == {17, 18, 65, 95} \* We model check 4 nodes system
MCA == {18} \* Initial READY node is 18
MCB == 4 \* base is 2^4
MCM == 8 \* ID space is 2^8
MCL == 1
LightNext == \E i, j \in I:
      \/ RouteLookup(i, j)
      \/ RouteJoinRequest(i, j)
      \/ Deliver(i, j)
      \/ ReceiveLSProbe(i)
      \/ ReceiveLSPrRpl(i)
      \/ ReceiveJoinRequest(i)
      \/ ReceiveJoinReply(i)
      \/ RequestLease(i)
      \/ ReceiveLReq(i)
      \/ ReceiveBLS(i)
      \*\/ Lookup(i, j)
      \*\/ Join(i, j)
      \/ LeaseExpired(i, j)
      \/ GrantExpired(i, j)
      \/ NodeLeft(i)
      \/ SuspectFaulty(i, j)
      \/ ProbeTimeOut(i, j)
      \/ DeclareDead(i, j)
      \/ LSRepair(i)
      \/ Recover(i)
      \/ Restart(i)
      \/ ResignNode(i)
----------------------
NodesOK    == {i \in I : status[i] = "ok"}
NodesReady == {i \in I : status[i] = "ready" }
ReadyOK    == NodesOK \cup NodesReady
----------------------------
MCALookup == {17, 18, 65}
LookupNext == \/ Lookup(18, 95)
              \/ LightNext
SpecSuccessLookup ==
/\ Init
/\ [][LookupNext]_vars
NeverDeliver == [][\A i, j \in I: ~Deliver(i, j)]_vars
-------------------------------------------------------------
JoinNext == \/ Join(95, 18)
            \/ LightNext
SpecSuccessJoin ==
/\ Init
/\ [][JoinNext]_vars
NeverJoin == [](\A j \in I \ A: status[j]#"ready")
------------------------------------------------------
ConcJoinNext == \/ Join(95, 18)
                \/ Join(17, 18)
                \/ LightNext
SpecConcJoin ==
/\ Init
/\ [][ConcJoinNext]_vars
ConcurrentJoin == []~(status[95] = "ready"
                          /\status[17] = "ready"
                          /\status[18]= "ready")
------------------------------------------------
CcJoinDeliverNext == \/ Join(95, 18)
                     \/ Join(17, 18)
                     \/ Lookup(95, 65)
                     \/ LightNext
```

```
SpecCcJoinDeliver ==
/\ Init
/\ [][CcJoinDeliverNext]_vars
CcJoinDeliver ==[][~\E i, j \in I: Deliver(i, j)
                          /\status[95]= "ready"
                          /\status[17]= "ready"
                          /\status[18]= "ready"]_vars
--------------------------------------------------------
SymmNext == \/ Join(95, 18)
            \/ Join(17, 18)
            \/ Join(65, 95)
            \/ LightNext
SpecSym ==
/\ Init
/\ [][SymmNext]_vars
Symmetry ==
\A i, j \in I:
   /\ status[i] = "ready"
   /\ status[j] = "ready"
   => (j \in GetLSetContent(lset[i])
           <=> i \in GetLSetContent(lset[j]))
-----------------------------------------------
MCANCF == {17, 18}
NCFNext == \/ Join(95, 17)
           \/ Join(65, 18)
           \/ LightNext
SpecNCF ==
/\ Init
/\ [][NCFNext]_vars
Neighbor ==
  \/ Cardinality(ReadyOK) =< 1
  \/ \A x, y \in ReadyOK:
     ~(x = y) =>
     /\ CwDist(LeftNeighbor(lset[x]), x)  =< CwDist(y, x)
     /\ CwDist(x, RightNeighbor(lset[x])) =< CwDist(x, y)
NeighborClosest ==
  \/ Cardinality(NodesReady) =< 1
  \/ \A x, y \in NodesReady:
     ~(x = y) =>
     /\ CwDist(LeftNeighbor(lset[x]), x)=< CwDist(y, x)
     /\ CwDist(x, RightNeighbor(lset[x])) =< CwDist(x, y)
```

```
HalfNeighbor ==
  \/ \A k \in ReadyOK :
      /\ RightNeighbor(lset[k]) # k
      /\ LeftNeighbor(lset[k]) # k
  \/ \E k \in ReadyOK:
     /\ ReadyOK = {k}
     /\ LeftNeighbor(lset[k]) = k
     /\ RightNeighbor(lset[k]) = k
NeighborProp == HalfNeighbor /\ NeighborClosest
------------------------------------------------------------
CorrectDelivery == \A i,k \in I:
  ENABLED Deliver(i, k)
  => /\ \A n \in I: status[n] = "ready"
             => AbsDist(i, k) =< AbsDist(n, k)
     /\ \A j \in I\{i}: ~ENABLED Deliver(j, k)
==================end of module MCMSPastry==================
------------Configuration File: MCMSPastry.cfg--------------
CONSTANTS
  I <- MCI
  A <- MCA
\*  A <- MCANCF
  B <- MCB
  M <- MCM
  L <- MCL
\*SPECIFICATION SpecSuccessLookup
\*PROPERTY NeverDeliver
\*SPECIFICATION SpecSuccessJoin
\*PROPERTY NeverJoin
\*SPECIFICATION SpecConcJoin
\*PROPERTY ConcurrentJoin
\*SPECIFICATION SpecCcJoinDeliver
\*PROPERTY CcJoinDeliver
SPECIFICATION SpecSym
\*INVARIANTS Symmetry
\*SPECIFICATION SpecNCF
\*INVARIANTS Neighbor
\*INVARIANTS NeighborClosest
\*INVARIANTS HalfNeighbor
\*INVARIANTS NeighborProp
PROPERTY CorrectDelivery
=====================end  configuration File==================
```

5 Theorem Proving

Having gained confidence in our model, we now turn to formally proving the main correctness Property 2, using the interactive TLA$^+$ proof system (TLAPS) [6]. Our full proofs are available on the Web[1].

The intuition gained from the counter-example of Section 3.4 tells us that the key to establishing Property 2 is that the leaf sets of all nodes participating in the protocol contain the expected elements. We start by defining a number of auxiliary formulas.

$$
\begin{array}{l}
\mathit{Ready} \triangleq \{i \in I : \mathit{status}[i] = \text{“ready”}\} \\
\mathit{ReadyOK} \triangleq \{i \in I : \mathit{status}[i] \in \{\text{“ready”}, \text{“ok”}\}\} \\
\mathit{HalfNeighbor} \triangleq \\
\quad \lor\ \forall i \in \mathit{ReadyOK} : \mathit{RightNeighbor}(\mathit{lset}[i]) \neq i \land \mathit{LeftNeighbor}(\mathit{lset}[i]) \neq i \\
\quad \lor\ \land\ \mathit{Cardinality}(\mathit{ReadyOK}) \leq 1 \\
\quad\quad \land\ \forall i \in \mathit{ReadyOK} : \mathit{LeftNeighbor}(\mathit{lset}[i]) = i \land \mathit{RightNeighbor}(\mathit{lset}[i]) = i \\
\mathit{NeighborClosest} \triangleq \forall i, j \in \mathit{Ready} : \\
\quad i \neq j \Rightarrow \land\ \mathit{CwDist}(\mathit{LeftNeighbor}(\mathit{lset}[i]), i) \leq \mathit{CwDist}(j, i) \\
\quad\quad\quad\quad\ \ \land\ \mathit{CwDist}(i, \mathit{RightNeighbor}(\mathit{lset}[i])) \leq \mathit{CwDist}(i, j)
\end{array}
$$

Sets *Ready* and *ReadyOK* contain the nodes that are “ready”, resp. “ready” or “ok”. Formula *HalfNeighbor* 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 particular, it follows by Definition 2 that no leaf set of i can be empty. The formula *NeighborClosest* states that the left and right neighbors of any “ready” node i lie closer to i than any “ready” node j different from i.

[1]The appendix at the end of this chapter shows the TLA$^+$ proof of essential lemmas illustrated below. There are hundreds of sub lemmas involved, which are properties of the data-structures, mathematic operations and ring calculations. All of them could be found in our full proofs available on the Web `http://www.mpi-inf.mpg.de/~tianlu/software/PastryTheoremProving.zip`

We used TLC to verify $\mathit{NeighborProp} \triangleq \mathit{HalfNeighbor} \land \mathit{NeighborClosest}$. Running TLC for more than a month did not yield a counter-example. Using TLAPS, we have mechanically proved that *NeighborProp* implies Property 2. In order to prove this, we have first proved the following two lemmas.

The lemma 1 shows that, assuming *NeighborProp*, then for any two "ready" nodes i, n, with $i \neq n$ and key k, if node i covers k then i must be at least as close to k as is n.

Lemma 1 (Coverage Lemma)**.**

$$\begin{aligned}&\mathit{HalfNeighbor} \land \mathit{NeighborClosest}\\ &\Rightarrow\ \forall i, n \in \mathit{Ready} : \forall k \in I : i \neq n \land \mathit{Covers}(\mathit{lset}[i], k)\\ &\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathit{AbsDist}(i, k) \leq \mathit{AbsDist}(n, k)\end{aligned}$$

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. There are two cases to consider: n and i could be on the same side of k, hence n lies between i and k, or nodes i and n could lie on opposite sides of the key k, but n is closer. Further distinguishing between the left and right neighborhoods of k, we obtain four cases, of which we sketch two (the others being symmetrical).

Suppose first that n and i are both to the left of k. Assumption *NeighborClosest* implies that the distance between i and its right neighbor rn is at most as large as the distance between i and n. Using the definition of *Cover* (Definition 2), we show that the clockwise distance from i to its right cover is strictly smaller than the clockwise distance from i to rn. By assumption *HalfNeighbor*, and since there are two "ready" nodes, we know that $rn \neq i$. Moreover, by the assumption that n is closer to k than i, and using again the definition of *Cover*, we prove 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 that to rn, obtaining a contradiction against our assumption at the beginning that rn is at most as far as n to i.

Now suppose that i is to the left and n to the right of k. Since i covers k, we know that 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, hence (by assumption *NeighborClosest*) at most half the distance between i and n. Therefore, k is closer to i than to n, and again a contradiction is reached. □

The lemma 2 shows, under the same hypotheses, that if i covers k then n cannot cover k.

Lemma 2 (Disjoint Covers)**.**

$$\begin{aligned}
&\mathit{HalfNeighbor} \land \mathit{NeighborClosest}\\
&\Rightarrow\ \forall i, n \in \mathit{Ready} : \forall k \in I : i \neq n \land \mathit{Covers}(\mathit{lset}[i], k)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \neg \mathit{Covers}(\mathit{lset}[n], k)
\end{aligned}$$

Proof. We prove the lemma by contradiction. Assume that both nodes i and n cover key k. By Lemma 1 it follows that the distances of i and n to k are the 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. Let rn denote the right neighbor of i (because i and n are both "ready", *HalfNeighbor* implies that $rn \neq i$). By assumption *NeighborClosest* we know that no "ready" node can lie between a node and its direct neighbor, so the distance from i to n must be at least that from i to rn. By the assumption that i covers k and the definition of coverage, the distance from i to rn is at least twice the distance from i to k, which is by the above just the distance from i to n, so we must have $rn = n$.

Now we have proved that n is the right neighbor of i. Since the asymmetrical definitions of *LeftCover* and *RightCover* imply that no two neighbors may cover a node, we then arrive at a contradiction against the assumption at the beginning of this proof.

□

Theorem 3 (Reduction)**.** $\mathit{HalfNeighbor} \land \mathit{NeighborClosest} \Rightarrow \mathit{CorrectDeliver}$.

Proof. Taking together Lemma 1 and Lemma 2, Theorem 3 follows easily by the definitions of the property *CorrectDeliver* (Property 2) and the action *Deliver* (cf. Figure 3.2). □

In order to complete the proof that our model of Pastry satisfies Property 2, it is enough by Theorem 3 to show that every reachable state satisfies properties *HalfNeighbor* and *NeighborClosest*. We have embarked on an invariant proof and have defined a predicate that strengthens these properties. We are currently in the process of showing that it is indeed preserved by all actions of our model.

```
----------------------MODULE ProofProp -------------------------
NodesOK    == {i \in I : status[i] = "ok"}
NodesReady == {i \in I : status[i] = "ready" }
NodesWait  == {i \in I : status[i] = "wait" }
ReadyOK    == NodesOK \union NodesReady
NonDead    == {i \in I : status[i] # "dead" }

THEOREM CoverageLemma ==
TypeInvariant /\  Invariant =>
   \A i, k, n \in I:
      /\ status[i] = "ready"
      /\ status[n] = "ready"
      /\ Covers(lset[i], k)
   => AbsDist(i, k) =< AbsDist(n, k)
<1>1.SUFFICES
           ASSUME NEW i \in I,
                  NEW k \in I,
                  NEW n \in I,
                  TypeInvariant,
                  Invariant,
                  status[i] = "ready",
                  status[n] = "ready",
                  Covers(lset[i], k),
                  ~(AbsDist(i, k) =< AbsDist(n, k))
           PROVE FALSE
   BY DEF Covers
<1>2. CASE n = i
   <2>1. AbsDist(i, k) = AbsDist(n, k)
      BY <1>2
   <2>2. ~(AbsDist(i, k) = AbsDist(n, k))
      BY <1>1, Unequality6, AbsDistType, IisNat
   <2>9. QED BY <2>1,<2>2
<1>3. CASE ~(n = i)
   <2>. lset[i] \in LSet
      BY TypeInvariant DEF TypeInvariant
   <2>01. ~RightNeighbor(lset[i]) = i /\~LeftNeighbor(lset[i]) = i
      BY status[i] = "ready", status[n] = "ready", <1>3,
           NoPartition, Invariant, TypeInvariant
   <2>1. CASE AbsDist(i, k) = CwDist(i, k) /\ AbsDist(n, k) = CwDist(n, k)
      <3>1. CwDist(n, k) < CwDist(i, k)
         BY <1>1, <2>1,  Unequality2, CwDistType, IisNat
      <3>3. CwDist(i, n) =< CwDist(i, k)
         BY <3>1, CwDistPropCo, IisNat
      <3>4. CwDist(i, k) =< CwDist(i, RightCover(lset[i]))
         BY <2>1, Covers(lset[i], k), CoverSemmanticCoRight, TypeInvariant
      <3>5. CwDist(i, n) =< CwDist(i, RightCover(lset[i]))
         BY <3>3, <3>4, TransLEQ, CwDistType, RightCoverType, IisNat
      <3>6. CwDist(i, RightCover(lset[i])) < CwDist(i, RightNeighbor(lset[i]))
         BY <1>3, NoPartition, <1>1, NeighborConventionRight, Invariant
      <3>7. CwDist(i, n) < CwDist(i, RightNeighbor(lset[i]))
         BY <3>5, <3>6, CwDistType, IisNat, RightNeighborType,
                         RightCoverType, TransLESS
      <3>8. ~(CwDist(i, n) < CwDist(i, RightNeighbor(lset[i])))
         BY status[i] = "ready", status[n] = "ready", Neighborhood,
                            StatusDuality, <1>3,
             CwDistType, RightNeighborType, IisNat, TypeInvariant, Invariant
      <3>9. QED BY <3>8, <3>7
   <2>2. CASE AbsDist(i, k) = CwDist(i, k) /\ AbsDist(n, k) = CwDist(k, n)
      <3>7. CwDist(i, k) > CwDist(k, n)
      BY ~(AbsDist(i, k) =< AbsDist(n, k)), <2>2,
         Unequality2, LessGreaterDuality, CwDistType, IisNat
      <3>8. CwDist(i, k) =< CwDist(k, n)
         <4>1. CwDist(i, n) = CwDist(i, k) + CwDist(k, n)
             <5>1. Distance(n, k) < 0
                BY <2>2, AbsDistRightSide, IisNat
             <5>2. Distance(k, i) < 0
                BY <2>2, AbsDistLeftSide, IisNat
             <5>4. CwDist(i, n) = CwDist(i, k) + CwDist(k, n)
                BY <1>3, <5>1, <5>2, RingAddProp, IisNat
             <5>9. QED BY <5>4
         <4>2. CwDist(i, k) =< CwDist(i, n) \div 2
             <5>1. CwDist(i, RightNeighbor(lset[i])) =< CwDist(i, n)
                <6>1. CASE RightNeighbor(lset[i]) = n
                   BY <6>1, Unequality6, IisNat,
                         RightNeighborType, CwDistType
                <6>2. CASE ~ (RightNeighbor(lset[i]) = n)
                   <7>2. ~CwDist(i, n) < CwDist(i, RightNeighbor(lset[i]))
                      BY <1>1, Neighborhood, StatusDuality, <1>3
                   <7>3. CwDist(i, n) >= CwDist(i, RightNeighbor(lset[i]))
                      BY <7>2, Unequality22, CwDistType,
                            IisNat, RightNeighborType
                   <7>9. QED BY <7>3, LessGreaterDuality2,
                        IisNat, RightNeighborType, CwDistType
                <6>9. QED BY <6>1, <6>2
              <5>2. CwDist(i, RightCover(lset[i]))
                        =< CwDist(i, RightNeighbor(lset[i])) \div 2
                BY <2>01, NeighborCoverageCoRight, IisNat,
                      RightCoverType, RightNeighborType, TypeInvariant
              <5>3. CwDist(i, RightCover(lset[i])) =< CwDist(i, n) \div 2
                <6>1. CwDist(i, RightNeighbor(lset[i])) \div 2
                                =< CwDist(i, n) \div 2
                  BY <5>1, <5>2, MonotonyDiv,CwDistType,
                                          IisNat, RightNeighborType
                <6>2. QED BY <6>1, <5>2, TransLEQ,
                DivType, CwDistType, RightNeighborType, RightCoverType,IisNat,
                TypeInvariant DEF TypeInvariant
              <5>4. CwDist(i, k)  =< CwDist(i, RightCover(lset[i]))
                BY <2>2, Covers(lset[i], k), CoverSemmanticCoRight,
                                      TypeInvariant
              <5>9. QED BY <5>4, <5>3, TransLEQ, CwDistType,
                                     RightCoverType, IisNat, DivType
          <4>9. QED BY <4>1, <4>2, Unequality5, CwDistType, IisNat
       <3>9. QED BY <3>7,<3>8, Unequality3, CwDistType, IisNat
    <2>3. CASE AbsDist(k, i) = CwDist(k, i) /\ AbsDist(k, n) = CwDist(k, n)
      \* i and n lie right to k
      \* Proof sketch: symmetry to the <2>1
        <3>1. CwDist(k, n) < CwDist(k, i)
           BY <2>3, <1>1, Unequality2, IisNat, CwDistType, AbsCommutativity
        <3>3. CwDist(n, i) =< CwDist(k, i)
           BY <3>1, CwDistProp, IisNat, SideDuality
        <3>4. CwDist(k, i) =< CwDist(LeftCover(lset[i]), i)
          BY <2>3, Covers(lset[i], k), CoverSemmanticCoLeft, TypeInvariant
        <3>5. CwDist(n, i) =< CwDist(LeftCover(lset[i]), i)
           BY <3>3, <3>4, TransLEQ, CwDistType, LeftCoverType, IisNat
        <3>6. CwDist(LeftCover(lset[i]), i) < CwDist(LeftNeighbor(lset[i]), i)
           BY <1>1, <1>3, NeighborConventionLeft, TypeInvariant, NoPartition
        <3>7. CwDist(n, i) < CwDist(LeftNeighbor(lset[i]), i)
           BY <3>5, <3>6, CwDistType, IisNat, LeftNeighborType,
                                 LeftCoverType, TransLESS
        <3>8. ~(CwDist(n, i) < CwDist(LeftNeighbor(lset[i]), i))
           BY status[i] = "ready", status[n] = "ready", Neighborhood,
                StatusDuality, <1>3, TypeInvariant, Invariant
        <3>9. QED BY <3>7,<3>8
```

```
<2>4. CASE AbsDist(i, k) = CwDist(k, i) /\ AbsDist(n, k) = CwDist(n, k)
   <3>7. CwDist(k, i) > CwDist(n, k)
   BY ~(AbsDist(i, k) =< AbsDist(n, k)), <2>4,
      Unequality2, LessGreaterDuality, CwDistType, IisNat
   <3>8. CwDist(k, i) =< CwDist(n, k)
      <4>1. CwDist(n, i) =  CwDist(n, k) + CwDist(k, i)
          <5>1. Distance(k, n) < 0
             BY <2>4, AbsDistLeftSide, IisNat
          <5>2. Distance(i, k) < 0
             BY <2>4, AbsDistRightSide, IisNat
          <5>4. CwDist(n, i) = CwDist(n, k) + CwDist(k, i)
             BY <5>1, <5>2, RingAddProp, IisNat, <1>3
          <5>9. QED BY <5>4
      <4>2. CwDist(k, i) =< CwDist(n, i) \div 2
          <5>1. CwDist(LeftNeighbor(lset[i]), i) =< CwDist(n, i)
             <6>1. CASE LeftNeighbor(lset[i]) = n
                BY <6>1, Unequality6, IisNat, LeftNeighborType, CwDistType
             <6>2. CASE ~ (LeftNeighbor(lset[i]) = n)
                <7>2. CwDist(n, i) >= CwDist(LeftNeighbor(lset[i]), i)
                   BY <1>1, Neighborhood, Unequality22, <1>3,
                   StatusDuality, CwDistType, IisNat, LeftNeighborType
                <7>9. QED BY <7>2, LessGreaterDuality2,
                            IisNat, LeftNeighborType, CwDistType
             <6>9. QED BY <6>1, <6>2
          <5>2. CwDist(LeftNeighbor(lset[i]), i) \div 2
                       = CwDist(LeftCover(lset[i]), i)
             BY <2>01, NeighborCoverageCoLeft, IisNat,
             LeftCoverType,LeftNeighborType, TypeInvariant
          <5>3. CwDist(LeftCover(lset[i]), i) =< CwDist(n, i) \div 2
             <6>1. CwDist(LeftNeighbor(lset[i]), i) \div 2
                         =< CwDist(n, i) \div 2
                BY <5>1, <5>2, MonotonyDiv, CwDistType,
                     IisNat, LeftNeighborType
             <6>2. QED BY <6>1, <5>2
          <5>4. CwDist(k, i)  =< CwDist(LeftCover(lset[i]), i)
             BY <2>4, Covers(lset[i], k), CoverSemmanticCoLeft,
                    TypeInvariant, AbsCommutativity, IisNat
          <5>9. QED BY <5>4, <5>3, TransLEQ, CwDistType,
                                 LeftCoverType, IisNat, DivType
      <4>9. QED BY <4>1, <4>2, CommutativityAdd, Unequality5,
            CwDistType, IisNat
      <3>9. QED BY <3>7,<3>8, Unequality3, CwDistType, IisNat
   <2>9. QED BY <2>1, <2>2, <2>3, <2>4, AbsIsLeftOrRight,
                           AbsCommutativity, IisNat
<1>9. QED BY <1>2, <1>3

THEOREM ConsistentInv ==
TypeInvariant /\ Invariant =>
\A x, y, k \in I:
   (   status[x] = "ready"
    /\ Covers(lset[x], k)
    /\ status[y] = "ready"
    /\ Covers(lset[y], k)
    )
    => x = y
<1>1. SUFFICES ASSUME NEW x \in I,
                      NEW y \in I,
                      NEW k \in I,
      status[x] = "ready",
      status[y] = "ready",
      Covers(lset[x], k),
      Covers(lset[y], k),
      TypeInvariant,
      Invariant,
      ~(x = y)
      PROVE FALSE
   BY DEF Covers
<1>. \A i \in I: lset[i] \in LSet
   BY TypeInvariant DEF TypeInvariant
<1>11. ~Covers(lset[x], y) /\ ~Covers(lset[y], x)
   BY <1>1, NotCovers
<1>2. AbsDist(x, k) =< AbsDist(y, k)
   BY <1>1, CoverageLemma
<1>3. AbsDist(y, k) =< AbsDist(x, k)
   BY <1>1, CoverageLemma
<1>4. AbsDist(x, k) = AbsDist(y, k)
   BY <1>2,<1>3, Antisymmetry, AbsDistType, IisNat
<1>5. CASE AbsDist(x, k) = CwDist(x, k) /\ AbsDist(y, k) = CwDist(y, k)
   BY <1>5, <1>4, CwDistInjectivity, ~(x = y), IisNat
<1>6. CASE AbsDist(x, k) = CwDist(x, k) /\ AbsDist(y, k) = CwDist(k, y)
   <2>1. CwDist(x, y) = CwDist(x, k) + CwDist(k, y)
      <3>1. Distance(k, x) < 0 /\ Distance(y, k) < 0
      BY <1>6, AbsDistRightSide, AbsDistLeftSide, IisNat
    <3>2. QED BY <3>1,  ~(x=y), RingAddProp, IisNat
  <2>2. RightNeighbor(lset[x]) = y
    <3>1. SUFFICES ASSUME NEW z \in I,
            RightNeighbor(lset[x]) = z,
            ~(z = y)
            PROVE FALSE
      BY IisNat, RightNeighborType
    <3>11. x # z
      BY <1>1, <3>1, NoPartition
    <3>2. CwDist(x, k) < CwDist(x, LeftCover(lset[z]))
      BY CoverDisjointnessCo, <3>11, RightNeighbor(lset[x]) = z, <1>6,
            Covers(lset[x], k), TypeInvariant
    <3>3. CwDist(x, z) < CwDist(x, y)
      <4>6.  ~(CwDist(x, z) = CwDist(x, y))
        BY ~(z = y), RightDistInjectivity, IisNat
      <4>8. ~(CwDist(x, z) > CwDist(x, y))
        <5>2. ~(CwDist(x, y) < CwDist(x, z))
          BY <1>1, Neighborhood, Invariant,
                  RightNeighbor(lset[x]) = z, StatusDuality
        <5>3. QED BY <5>2, LessGreaterDuality, CwDistType, IisNat
      <4>9. QED BY <4>6, <4>8, Unequality23, CwDistType, IisNat
    <3>4. CwDist(x, LeftCover(lset[z])) =< CwDist(x, LeftCover(lset[y]))
      BY CoverMonotony, <3>3, RightNeighbor(lset[x]) = z,
      status[x] = "ready", status[y] = "ready", ~x=y, z#y,
        TypeInvariant, Invariant
    <3>5. CwDist(x, k) < CwDist(x, LeftCover(lset[y]))
      BY <3>2,<3>4,TransLESS, CwDistType, LeftCoverType, IisNat
    <3>6. CwDist(y, k) > CwDist(y, RightCover(lset[y]))
      <4>7. CwDist(y, k) >= CwDist(y, x)
        BY CwDistSideProp, <1>6, IisNat, LessGreaterDuality2
      <4>8. CwDist(y, x) > CwDist(y, RightCover(lset[y]))
        BY <1>11, CoverSemmanticCo, TypeInvariant
      <4>9. QED BY <4>7, <4>8, TransGEATER, CwDistType,
              RightCoverType, IisNat
    <3>7. CwDist(k, y) > CwDist(LeftCover(lset[y]), y)
      BY <3>5, <1>11, CoverSemmanticCo, CwDistPropAddCo,
      LeftCoverType, CwDistType, IisNat, TypeInvariant
    <3>8. ~Covers(lset[y], k)
      BY <3>7, <3>6, CoverSemmanticCo, TypeInvariant
    <3>9. QED BY <3>8, <1>1
```

```
   <2>10. QED BY <1>1, <2>2, RightNeighborCoverageDis
<1>7. CASE AbsDist(x, k) = CwDist(k, x) /\ AbsDist(y, k) = CwDist(y, k)
   \* symmetric to <1>6, simply switch x and y, change <1>6 to <1>7.
   \* complete proof see
   \* http://www.mpi-inf.mpg.de/~tianlu/software/PastryTheoremProving.zip
      OMITTED
<1>8. CASE AbsDist(x, k) = CwDist(k, x) /\ AbsDist(y, k) = CwDist(k, y)
\* symmetric to <1>5
   BY <1>8, <1>4, RightDistInjectivity, ~(x = y), IisNat
<1>9. QED BY <1>5, <1>6,<1>7,<1>8, AbsIsLeftOrRight

THEOREM CoverDisjoint ==
   TypeInvariant/\ Invariant => \A i, j, k \in I:
      /\status[i]= "ready"
      /\status[j]= "ready"
      /\ Covers(lset[i], k)
      /\ ~(j=i)
    =>  ~Covers(lset[j], k)
BY ConsistentInv
============end of main reduction proof in module ProofProp=============
```

6 Conclusion and Future Work

In this report we have presented a formal model of the Pastry routing protocol, a fundamental building block of P2P overlay networks. To the best of our knowledge, this is the first formal model of Pastry, although the application of formal modeling and verification techniques to P2P protocols is not entirely new. For example, Velipalasar et al. [16] report on experiments of applying the Spin model checker to a model of a communication protocol used in a P2P multimedia system. More closely related to our topic, Borgström et al. [2] present initial work toward the verification of a distributed hash table in a P2P overlay network in a process calculus setting, but only considered fixed configurations with perfect routing information. As we have seen, the main challenge in verifying Pastry lies in the correct handling of nodes joining the system on the fly. Bakhshi and Gurov [1] model the Pure Join protocol of Chord in the π-calculus and show that the routing information along the ring eventually stabilizes in the presence of potentially concurrent joins. Numerous technical differences aside, they do not consider possible interferences between the join and lookup sub-protocols, as we do in our model.

Pastry is a reasonably complex algorithm that mixes complex data structures, dynamic network protocols, and timed behavior for periodic node updates. We decided to abstract from timing aspects, which are mainly important for performance, but otherwise model the algorithm as faithfully as possible. Our main difficulties were to fill in details that are not obvious from the published descriptions of the algorithm, and to formally state the correctness properties expected from Pastry. In this respect, the model checker helped us understand the need for the extension of the join protocol by lease granting, as described in [8]. It was also invaluable to improve our understanding of the protocol because it allowed us to state "what-if" questions and refute conjectures such as the symmetry of leaf set membership (Property 3). The building of the first overall model of Pastry in TLA$^+$ took us about 3 months. Almost two third of it was devoted to the formal development of the underlying data structures, such as the address ring, leaf sets or routing tables.

After having built up confidence in the correctness of our model, we started full formal verification using theorem proving. In particular, we have reduced the correctness Property 2 to a predicate about leaf sets that the algorithm should maintain, and have defined a candidate for an inductive invariant. Future work will include full verification of the correctness properties. Afterward, we will extend the model by considering liveness properties, which obviously require assumptions about the ring being sufficiently stable. We also intend to study which parts of the proof are amenable to automated theorem proving techniques, as the effort currently required by interactive proofs is too high to scale to more complete P2P protocols.

Bibliography

[1] R. Bakhshi and D. Gurov. Verification of peer-to-peer algorithms: A case study. *Electr. Notes Theor. Comput. Sci.*, 181:35–47, 2007.

[2] J. Borgström, U. Nestmann, L. O. Alima, and D. Gurov. Verifying a structured peer-to-peer overlay network: The static case. In Priami and Quaglia [12], pages 250–265.

[3] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual ring routing: network routing inspired by DHTs. *SIGCOMM Comput. Commun. Rev.*, 36(4):351–362, 2006.

[4] M. Castro, M. Costa, and A. 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.

[5] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. A TLA$^+$ proof system. In *Workshop on Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA)*, volume CEUR Workshop Proceedings 418, pages 17–37, 2008.

[6] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA$^+$ proof system. In Giesl and Hähnle [7], pages 142–148.

[7] J. Giesl and R. Hähnle, editors. *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*. Springer, 2010.

[8] A. Haeberlen, J. Hoye, A. Mislove, and P. Druschel. Consistent key mapping in structured overlays. Technical Report TR05-456, Rice University, Department of Computer Science, August 2005.

[9] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

[10] T. Lu, S. Merz, and C. Weidenbach. Towards verification of the Pastry routing protocol using TLA+. In R. Bruni and J. Dingel, editors, *Proceeding of IFIP International Conference on Formal Techniques for Distributed Systems (FORTE 2011)*, Reykjavik, Iceland, 2011. Springer. Accepted.

[11] S. Merz. The specification language TLA$^+$. In D. Bjrner and M. C. Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science, pages 401–451. Springer, Berlin-Heidelberg, 2008.

[12] C. Priami and P. Quaglia, editors. *Global Computing, IST/FET International Workshop, GC 2004, Rovereto, Italy, March 9-12, 2004, Revised Selected Papers*, volume 3267 of *Lecture Notes in Computer Science.* Springer, 2005.

[13] Rice University and Max-Planck Institute for Software Systems. Pastry: A substrate for peer-to-peer applications. `http://www.freepastry.org/`.

[14] R. Rodrigues and P. Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, 2010.

[15] A. Rowstron and P. 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, Nov. 2001.

[16] S. Velipasalar, C. H. Lin, J. Schlessman, and W. Wolf. Design and verification of communication protocols for peer-to-peer multimedia systems. In *IEEE Intl. Conf. Multimedia and Expo (ICME 2006)*, pages 1421–1424, Toronto, Canada, 2006. IEEE.

[17] Y. Yu, P. Manolios, and L. 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.

MPI-I-2012-RG1-001	M. Suda, C. Weidenbach	Labelled superposition for PLTL
MPI-I-2011-5-002	B. Taneva, M. Kacimi, G. Weikum	Finding images of rare and ambiguous entities
MPI-I-2011-4-005	A. Berner, O. Burghard, M. Wand, N.J. Mitra, R. Klein, H. Seidel	A morphable part model for shape manipulation
MPI-I-2011-4-001	M. Granados, J. Tompkin, K. In Kim, O. Grau, J. Kautz, C. Theobalt	How not to be seen inpainting dynamic objects in crowded scenes
MPI-I-2010-RG1-001	M. Suda, C. Weidenbach, P. Wischnewski	On the saturation of YAGO
MPI-I-2010-5-008	S. Elbassuoni, M. Ramanath, G. Weikum	Query relaxation for entity-relationship search
MPI-I-2010-5-007	J. Hoffart, F.M. Suchanek, K. Berberich, G. Weikum	YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia
MPI-I-2010-5-006	A. Broschart, R. Schenkel	Real-time text queries with tunable term pair indexes
MPI-I-2010-5-005	S. Seufert, S. Bedathur, J. Mestre, G. Weikum	Bonsai: Growing Interesting Small Trees
MPI-I-2010-5-004	N. Preda, F. Suchanek, W. Yuan, G. Weikum	Query evaluation with asymmetric web services
MPI-I-2010-5-003	A. Anand, S. Bedathur, K. Berberich, R. Schenkel	Efficient temporal keyword queries over versioned text
MPI-I-2010-5-002	M. Theobald, M. Sozio, F. Suchanek, N. Nakashole	URDF: Efficient Reasoning in Uncertain RDF Knowledge Bases with Soft and Hard Rules
MPI-I-2010-5-001	K. Berberich, S. Bedathur, O. Alonso, G. Weikum	A language modeling approach for temporal information needs
MPI-I-2010-1-001	C. Huang, T. Kavitha	Maximum cfardinality popular matchings in strict two-sided preference lists
MPI-I-2009-RG1-005	M. Horbach, C. Weidenbach	Superposition for fixed domains
MPI-I-2009-RG1-004	M. Horbach, C. Weidenbach	Decidability results for saturation-based model building
MPI-I-2009-RG1-002	P. Wischnewski, C. Weidenbach	Contextual rewriting
MPI-I-2009-RG1-001	M. Horbach, C. Weidenbach	Deciding the inductive validity of $\forall\exists^*$ queries
MPI-I-2009-5-007	G. Kasneci, G. Weikum, S. Elbassuoni	MING: Mining Informative Entity-Relationship Subgraphs
MPI-I-2009-5-006	S. Bedathur, K. Berberich, J. Dittrich, N. Mamoulis, G. Weikum	Scalable phrase mining for ad-hoc text analytics
MPI-I-2009-5-005	G. de Melo, G. Weikum	Towards a Universal Wordnet by learning from combined evidenc
MPI-I-2009-5-004	N. Preda, F.M. Suchanek, G. Kasneci, T. Neumann, G. Weikum	Coupling knowledge bases and web services for active knowledge
MPI-I-2009-5-003	T. Neumann, G. Weikum	The RDF-3X engine for scalable management of RDF data
MPI-I-2009-5-002	M. Ramanath, K.S. Kumar, G. Ifrim	Generating concise and readable summaries of XML documents
MPI-I-2009-4-006	C. Stoll	Optical reconstruction of detailed animatable human body models

MPI-I-2009-4-005	A. Berner, M. Bokeloh, M. Wand, A. Schilling, H. Seidel	Generalized intrinsic symmetry detection
MPI-I-2009-4-004	V. Havran, J. Zajac, J. Drahokoupil, H. Seidel	MPI Informatics building model as data for your research
MPI-I-2009-4-003	M. Fuchs, T. Chen, O. Wang, R. Raskar, H.P.A. Lensch, H. Seidel	A shaped temporal filter camera
MPI-I-2009-4-002	A. Tevs, M. Wand, I. Ihrke, H. Seidel	A Bayesian approach to manifold topology reconstruction
MPI-I-2009-4-001	M.B. Hullin, B. Ajdin, J. Hanika, H. Seidel, J. Kautz, H.P.A. Lensch	Acquisition and analysis of bispectral bidirectional reflectance distribution functions
MPI-I-2008-RG1-001	A. Fietzke, C. Weidenbach	Labelled splitting
MPI-I-2008-5-004	F. Suchanek, M. Sozio, G. Weikum	SOFI: a self-organizing framework for information extraction
MPI-I-2008-5-003	G. de Melo, F.M. Suchanek, A. Pease	Integrating Yago into the suggested upper merged ontology
MPI-I-2008-5-002	T. Neumann, G. Moerkotte	Single phase construction of optimal DAG-structured QEPs
MPI-I-2008-5-001	G. Kasneci, M. Ramanath, M. Sozio, F.M. Suchanek, G. Weikum	STAR: Steiner tree approximation in relationship-graphs
MPI-I-2008-4-003	T. Schultz, H. Theisel, H. Seidel	Crease surfaces: from theory to extraction and application to diffusion tensor MRI
MPI-I-2008-4-002	D. Wang, A. Belyaev, W. Saleem, H. Seidel	Estimating complexity of 3D shapes using view similarity
MPI-I-2008-1-001	D. Ajwani, I. Malinger, U. Meyer, S. Toledo	Characterizing the performance of Flash memory storage devices and its impact on algorithm design
MPI-I-2007-RG1-002	T. Hillenbrand, C. Weidenbach	Superposition for finite domains
MPI-I-2007-5-003	F.M. Suchanek, G. Kasneci, G. Weikum	Yago : a large ontology from Wikipedia and WordNet
MPI-I-2007-5-002	K. Berberich, S. Bedathur, T. Neumann, G. Weikum	A time machine for text search
MPI-I-2007-5-001	G. Kasneci, F.M. Suchanek, G. Ifrim, M. Ramanath, G. Weikum	NAGA: searching and ranking knowledge
MPI-I-2007-4-008	J. Gall, T. Brox, B. Rosenhahn, H. Seidel	Global stochastic optimization for robust and accurate human motion capture
MPI-I-2007-4-007	R. Herzog, V. Havran, K. Myszkowski, H. Seidel	Global illumination using photon ray splatting
MPI-I-2007-4-006	C. Dyken, G. Ziegler, C. Theobalt, H. Seidel	GPU marching cubes on shader model 3.0 and 4.0
MPI-I-2007-4-005	T. Schultz, J. Weickert, H. Seidel	A higher-order structure tensor
MPI-I-2007-4-004	C. Stoll, E. de Aguiar, C. Theobalt, H. Seidel	A volumetric approach to interactive shape editing
MPI-I-2007-4-003	R. Bargmann, V. Blanz, H. Seidel	A nonlinear viseme model for triphone-based speech synthesis
MPI-I-2007-4-002	T. Langer, H. Seidel	Construction of smooth maps with mean value coordinates
MPI-I-2007-4-001	J. Gall, B. Rosenhahn, H. Seidel	Clustered stochastic optimization for object recognition and pose estimation
MPI-I-2007-2-001	A. Podelski, S. Wagner	A method and a tool for automatic veriication of region stability for hybrid systems
MPI-I-2007-1-003	A. Gidenstam, M. Papatriantafilou	LFthreads: a lock-free thread library
MPI-I-2007-1-002	E. Althaus, S. Canzar	A Lagrangian relaxation approach for the multiple sequence alignment problem
MPI-I-2007-1-001	E. Berberich, L. Kettner	Linear-time reordering in a sweep-line algorithm for algebraic curves intersecting in a common point
MPI-I-2006-5-006	G. Kasnec, F.M. Suchanek, G. Weikum	Yago - a core of semantic knowledge