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

Tianxiang Lu[1,2], Stephan Merz[2], and Christoph Weidenbach[1]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
{tianlu,weidenbach}@mpi-inf.mpg.de
[2] INRIA Nancy & LORIA, Nancy, France
Stephan.Merz@loria.fr

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

1 Introduction

Pastry [9, 3, 5] 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.

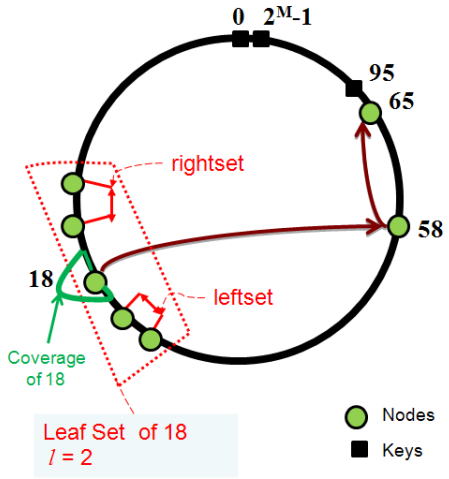

Fig. 1. Pastry Routing Example

Pastry routes a message by forwarding it to nodes that match progressively longer prefixes with the destination key. In the example of Fig. 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 Fig. 4).

The first challenge in modeling Pastry was to determine an appropriate level of abstraction. As a guiding principle, we focused the model towards 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 modelled 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 [8]. 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 [9, 3, 5]. 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 paper 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. 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 6. The paper ends with a summary of our results, related work, and future directions of research in Section 6. Further details and all proofs can be found in a technical report [7].

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 presents a first version of the join protocol in more detail, according to our understanding from [9] and [3]. We will refine this protocol in Sect. 4 according to the description in [5].

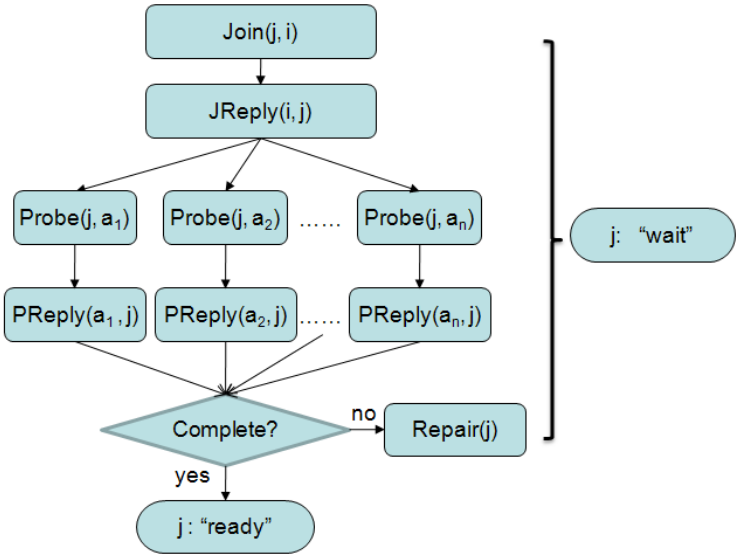

Fig. 2. Overview of the join protocol.

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 Fig. 2 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$^+$ [6]. Although there are of course alternative logics and respective theorem provers for modelling 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[3]. 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 $x, y \in I$:*

$$Dist(x,y) \triangleq \begin{cases} x - y + 2^{M-1} & \text{if } x - y < -2^{M-1} \\ x - y - 2^{M-1} & \text{if } x - y > 2^{M-1} \\ x - y, & \text{else} \end{cases}$$

$$AbsDist(x,y) \triangleq |Dist(x,y)|$$

$$CwDist(x,y) \triangleq \begin{cases} AbsDist(x,y) & \text{if } Dist(x,y) < 0 \\ 2^M - AbsDist(x,y) & \text{else} \end{cases}$$

The sign of $Dist(x, y)$ 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 $AbsDist(x, y)$ gives the length of the shortest path along the ring from x to y. Finally, the clockwise distance $CwDist(x, y)$ returns the length of the clockwise path from x to y.

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.

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

Definition 2 (Operations on Leaf Sets).

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

$$LeftNeighbor(ls) \triangleq \begin{cases} ls.node & \text{if } ls.left = \{\} \\ n \in ls.left : \forall p \in ls.left : & \\ \quad CwDist(p, ls.node) & \\ \quad \geq CwDist(n, ls.node) & \text{else} \end{cases}$$

$$RightNeighbor(ls) \triangleq \begin{cases} ls.node & \text{if } ls.right = \{\} \\ n \in ls.right : \forall q \in ls.right : & \\ \quad CwDist(ls.node, q) & \\ \quad \geq CwDist(ls.node, n) & \text{else} \end{cases}$$

$$LeftCover(ls) \triangleq (ls.node + CwDist(LeftNeighbor(ls), ls.node) \div 2)\%2^M$$

$$RightCover(ls) \triangleq (RightNeighbor(ls) + CwDist(ls.node, RightNeighbor(ls)) \div 2 + 1)\%2^M$$

$$Covers(ls, k) \triangleq CwDist(LeftCover(ls), k) \leq CwDist(LeftCover(ls), RightCover(ls))$$

In these definitions, $\div$ and $\%$ stand for division and modulo on the natural numbers, respectively. Note that they are in fact only applied in the above definitions to natural numbers. We also define the operation $AddToLSet(A, ls)$ 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.

3.2 Dynamic Model

Fig. 3 shows the high-level outline of the transition model specification in TLA$^+$. The overall system specification *Spec* is defined as $Init \wedge \Box[Next]_{vars}$, 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

$$
\begin{array}{lcl}
\mathit{vars} & \triangleq & \langle \mathit{receivedMsgs}, \mathit{status}, \mathit{lset}, \mathit{probing}, \mathit{failed}, \mathit{rtable} \rangle \\
\mathit{Init} & \triangleq & \land\ \mathit{receivedMsgs} = \{\} \\
 & & \land\ \mathit{status} = [i \in I \mapsto \text{IF } i \in A \text{ THEN “ready” ELSE “dead”}] \\
 & & \land\ \mathit{lset} = [i \in I \mapsto \text{IF } i \in A \\
 & & \qquad\qquad \text{THEN } \mathit{AddToLSet}(A, [\mathit{node} \mapsto i, \mathit{left} \mapsto \{\}, \mathit{right} \mapsto \{\}]) \\
 & & \qquad\qquad \text{ELSE } [\mathit{node} \mapsto i, \mathit{left} \mapsto \{\}, \mathit{right} \mapsto \{\}]] \\
 & & \land\ \mathit{probing} = [i \in I \mapsto \{\}] \\
 & & \land\ \mathit{failed} = [i \in I \mapsto \{\}] \\
 & & \land\ \mathit{rtable} = \ldots \\
\mathit{Next} & \triangleq & \exists i, j \in I : \lor\ \mathit{Deliver}(i, j) \\
 & & \qquad\qquad \lor\ \mathit{Join}(i, j) \\
 & & \qquad\qquad \lor\ \mathit{JReply}(i, j) \\
 & & \qquad\qquad \lor\ \mathit{Probe}(i, j) \\
 & & \qquad\qquad \lor\ \mathit{PReply}(i, j) \\
 & & \qquad\qquad \lor\ \ldots \\
\mathit{Spec} & \triangleq & \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}}
\end{array}
$$

Fig. 3. Overall Structure of the TLA$^+$ Specification of Pastry.

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 its routing table. The predicate *Init* is defined as a conjunction that initializes all variables; in particular, the model takes a parameter A indicating the set of nodes that are initially “ready”.

The next-state relation *Next* is a disjunction of all possible system actions, for all pairs of identifiers $i, j \in I$. Each action is defined as a TLA$^+$ action formula, 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, Fig. 4 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

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

Fig. 4. TLA$^+$ specification of action *Deliver*.

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 $e' = e$).

3.3 Validation By Model Checking

We used TLC [11], 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 3 (NeverDeliver and NeverJoin)

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

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 4 (Correct Key Delivery)

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

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, *CorrectDeliver* 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]

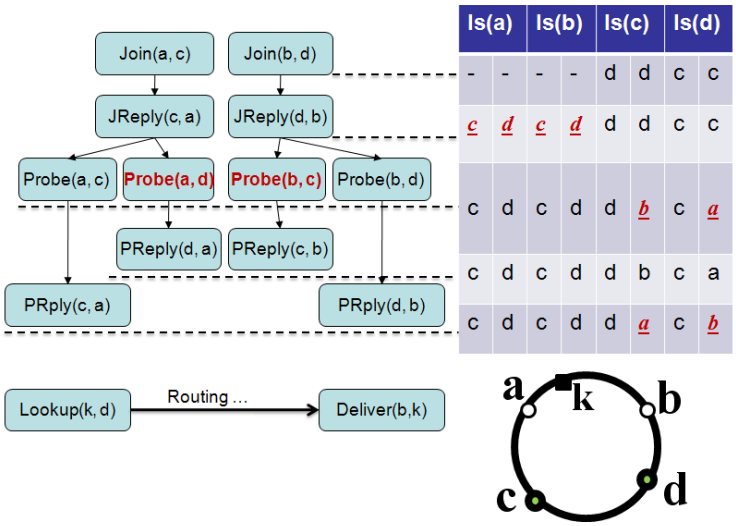

Fig. 5. Counter-example leading to a violation of *CorrectDeliver*.

When we attempted to verify Property 4, the model checker produced a counter-example, which we illustrate in Fig. 5. 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. Fig. 5), 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,

[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.

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. [5] 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 [5], 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 Fig. 5, 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 modelled it, and our further verification efforts. In fact, our formal model is also inspired by the implementation in FreePastry [8], where nodes periodically exchange their leaf sets to spread information about nodes dropping off and arriving.

4.1 Lease Granting Protocol

Figure 6 depicts the extension to the join protocol as described in Section 2 (cf. Fig. 2). 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".

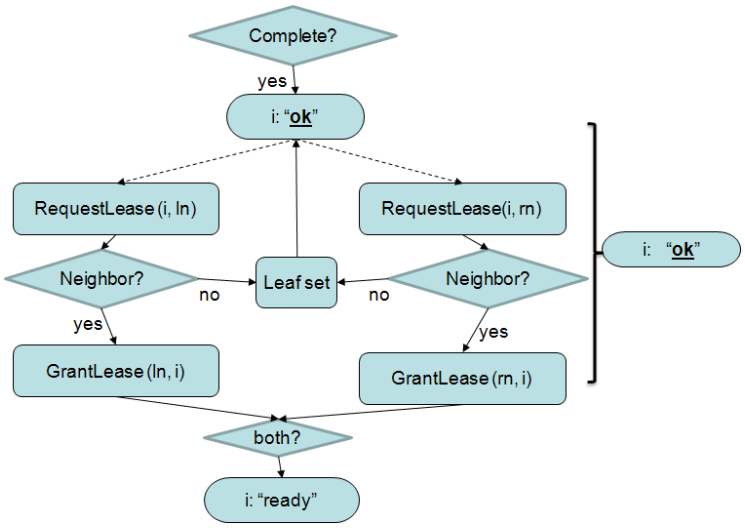

Fig. 6. Extending the Join Protocol by Lease Granting.

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.

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 4. However, we were unable to complete the model checking run and killed TLC after it had been running for more than a month.

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 5 (Symmetry of leaf set membership)

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

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

Examples	Time	Depth	# states	Counter Example
NeverDeliver	1"	5	101	yes
NeverJoin	1"	9	19	yes
ConcurrentJoin	3'53"	21	212719	yes
CcJoinLookup	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 1. TLC result with four nodes, leaf set length $l = 1$

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 1 summarizes the model checking experiments we 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 4, until we killed the process), and the largest depth of these states. Since the verification of Property 4 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 $\mathit{HalfNeighbor} \wedge \mathit{NeighborClosest}$, see the next section.

5 Theorem Proving

Having gained confidence in our model, we now turn to formally proving the main correctness Property 4, using the interactive TLA$^+$ proof system (TLAPS) [4].

Our full proofs are available on the Web[5]. The intuition gained from the counter-example of Section 3.4 tells us that the key to establishing Property 4 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.

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

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.

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 4, as asserted by the following theorem.

Theorem 6 (Reduction). $\mathit{HalfNeighbor} \land \mathit{NeighborClosest} \Rightarrow \mathit{CorrectDeliver}$.

We sketch our mechanically verified TLAPS proof of Theorem 6 by two lemmas. The first lemma shows that, assuming the hypotheses of Theorem 6, 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 7 (Coverage Lemma).

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

The second lemma shows, under the same hypotheses, that if i covers k then n cannot cover k. Taking together Lemma 7 and Lemma 8, Theorem 6 follows easily by the definitions of the property *CorrectDeliver* (Property 4) and the action *Deliver* (see Fig. 4).

[5] `http://www.mpi-inf.mpg.de/~tianlu/software/PastryTheoremProving.zip`

Lemma 8 (Disjoint Covers).

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

In order to complete the proof that our model of Pastry satisfies Property 4, it is enough by Theorem 6 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.

6 Conclusion and Future Work

In this paper 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. [10] 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 towards 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 [5]. 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 5). 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 4 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. Afterwards, 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.

Acknowledgements: We would like to thank the reviewers for their valuable comments.

References

1. Rana Bakhshi and Dilian Gurov. Verification of peer-to-peer algorithms: A case study. *Electr. Notes Theor. Comput. Sci.*, 181:35–47, 2007.
2. Johannes Borgström, Uwe Nestmann, Luc Onana Alima, and Dilian Gurov. Verifying a structured peer-to-peer overlay network: The static case. In Corrado Priami and Paola Quaglia, editors, *Global Computing*, volume 3267 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2004.
3. Miguel Castro, Manuel Costa, and Antony I. T. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *International Conference on Dependable Systems and Networks (DSN 2004)*, pages 9–18, Florence, Italy, 2004. IEEE Computer Society.
4. Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and StephanMerz. Verifying safety properties with the TLA+ proof system. In Jürgen Giesl and Reiner Hähnle, editors, *Intl. Joint Conf. Automated Reasoning (IJCAR 2010)*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148. Springer, 2010.
5. Andreas Haeberlen, Jeff Hoye, Alan Mislove, and Peter Druschel. Consistent key mapping in structured overlays. Technical Report TR05-456, Rice University, Department of Computer Science, August 2005.
6. Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
7. Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Towards verification of the pastry protocol using TLA+. Technical Report MPI-I-2011-RG1-002, Max-Planck-Institute fr Informatik, April 2011.
8. Rice University and Max-Planck Institute for Software Systems. Pastry: A substrate for peer-to-peer applications. `http://www.freepastry.org/`.
9. Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.
10. Senem Velipasalar, Chang Hong Lin, Jason Schlessman, and Wayne 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.
11. Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer.