Formal Verification of the Pastry Protocol Using TLA$^+$

Tianxiang Lu(✉)

Department of Computer Science,
Technische Universität Darmstadt, Darmstadt, Germany
contact@tiit.lu

Abstract. As a consequence of the rise of cloud computing, the reliability of network protocols is gaining increasing attention. However, formal methods have revealed inconsistencies in some of these protocols, e.g., Chord, where all published versions of the protocol have been discovered to be incorrect. Pastry is a protocol similar to Chord. Using TLA$^+$, a formal specification language, we show that LuPastry, a formal model of Pastry with some improvements, provides correct delivery service. This is the first formal proof of Pastry where concurrent joins and lookups are simultaneously allowed. In particular, this article relaxes the assumption from previous publication to allow arbitrary concurrent joins of nodes, which reveals new insights into Pastry through a final formal model in TLA$^+$, LuPastry. Besides, this article also illustrates the methodology for the discovery and proof of its invariant. The proof in TLA$^+$ is mechanically verified using the interactive theorem prover TLAPS.

Keywords: Formal verification · Interactive theorem proving · Network protocols

1 Introduction

1.1 The Pastry Protocol

Pastry ([16], [3], [4]) is a structured *P2P* algorithm realizing a Distributed Hash Table (*DHT*, by [5]) over an underlying virtual ring. 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[1] as shown in Fig. 1, i.e. $2^M - 1$ is the neighbor of 0.

The IDs are also used as object keys, such that an overlay node is responsible for keys that are numerically close to its ID, i.e. it provides the primary storage for the hash table entries associated with these keys. Key responsibility is divided equally according to the distance between two adjacent 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.

[1] The ring here does not refer to algebraic group structure with operation.

X. Li et al. (Eds.): SETTA 2015, LNCS 9409, pp. 284–299, 2015.
DOI: 10.1007/978-3-319-25942-0_19

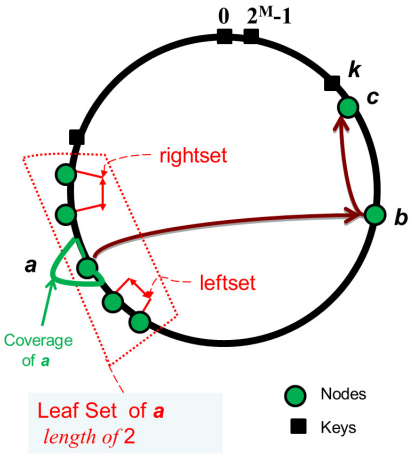

Fig. 1. Pastry ring.

The lookup protocol delivers the hash table entry for a given key. This paper focuses on the correctness property *CorrectDelivery* (mentioned as dependability in algorithm paper [3]), 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 concurrent join or departure of nodes, i.e., *churn*. To cope with that, each Pastry node maintains a local state of a set of nodes called *leaf sets*, as shown in Fig. 1, consisting of a left set and a right set of the same length, which is a parameter of the algorithm. The nodes in leaf sets are updated when new nodes join or failed nodes are detected using a maintenance protocol. A Pastry node also maintains a routing table to store more distant nodes, in order to achieve efficient routing.

In the example of Fig. 1, node a received a lookup message for key k. The key is outside node a's coverage. Moreover, it doesn't lie between the leftmost node and the rightmost node of its leaf sets. Querying its routing table, node a finds node b, whose identifier matches the longest prefix with the destination key and then forwards the message to that node. Node b repeats the process and finally, the lookup message is answered by node c, which covers the key k. In this case, we say that node c *delivers* the lookup request for key k.

1.2 The Methodology

TLA$^+$ by [6], is a formal specification language based on untyped *Zermelo-Fraenkel* (ZF) set theory with choice for specifying data structures, and on the Temporal Logic of Actions (TLA) for describing system behavior. It is chosen to analyze and verify the correct delivering and routing functionality of Pastry, because it provides a uniform logic framework for specification, model-checking and theorem proving. It fits protocol verification quite nicely, because its concept of actions matches the rule/message-based definition of protocols. In addition,

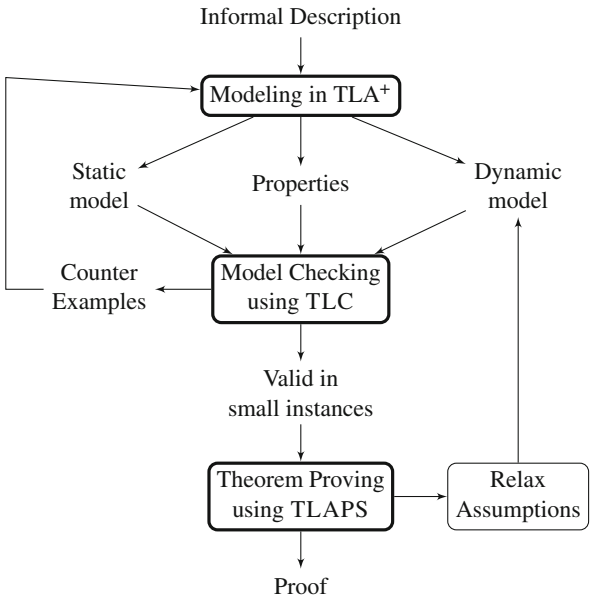

Fig. 2. Verification approach using TLA$^+$.

the specification language is straightforward to understand with basic mathematics and classical first-order logic. Furthermore, the convenient toolbox available in [7] includes now both the TLC model checker and the TLAPS proof system.

Fig. 2 illustrates the complete process of this framework which includes modeling, model checking and theorem proving.

Starting with an informal description of Pastry in [3], the first task is to model the requirements of the system using TLA$^+$. This paper distinguishes different kinds of TLA$^+$ model as *properties* that specify requirements using logic formulas (e.g. *CorrectDelivery*), the *static model* that defines the data structures (e.g. the virtual ring of IDs, the leaf sets etc.), and the *dynamic model* that describes the behavior using actions of TLA$^+$.

The challenges here include modeling Pastry on an appropriate level of abstraction, filling in needed details in the formal model that are not contained in the published description of Pastry, and formulating the correctness property of Pastry. These challenges motivate a deeper understanding of the protocol using the model checker TLC.

The model is validated using TLC in an iterative process, which helped to discover unexpected corner cases to improve the model and to ensure that the system has at least some useful executions. For example, accessibility properties are model-checked by checking that the negation is false. To avoid the state explosion problem, only a restricted number of instances are verified using TLC. Upon finding counterexamples, the model is analyzed and reformulated.

Upon successful validation or in the absence of counterexamples after running it for a considerable long time, the model is then verified by TLA$^+$ proofs.

Since the previous publications [11] and [12] have already included the model checking result, we omit it here to save space for illustrating theorem proving details.

The proof of the Pastry join protocol contains an induction part, where invariants need to be found and formulated. Discovering the invariants is the most subtle part of the theorem proving approach. On the one hand, it should be general enough to imply the target property *CorrectDelivery*. On the other hand, it should be specific enough to be provable using itself as induction hypothesis. In order to prove the safety property *CorrectDelivery*, we assume no departure of nodes and that an active node can help at most one node to join at a time, which relaxes the previous assumption we made in [13].

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

2 Modelling the Concurrent Join Protocol of Pastry

As illustrated in Fig. 2, the formal model of Pastry consists of a static part specifying the underlying data structures and a dynamic part specifying the behaviour of the nodes. Due to the page limit for this paper, the formal description of all the dynamic actions [10] are omitted here.

2.1 Static Model

The static model of Pastry consists of definitions of data structures and operations on them. A data structure is always a boolean value, a natural number, a set, a function or a complex composition of them. An operation on a data structure is always a functional mapping from a given signature of data structures to a returned value, which is again a data structure.

The static model of Pastry remains the same as in [12]. The distance between two nodes on the ring is modeled with clockwise distance $CwDist(x, y)$, which returns how many identifiers lie within the region from node x to y in the clockwise direction. The absolute value $AbsDist(x, y)$ gives the length of the shortest path along the ring from x to y. The routing table has a rather complex data structure, which is used for efficient routing. Since this article focuses on the safety property, details of the routing table are omitted here. The simplified routing table is a set of nodes ($RTable \triangleq [I \to \text{SUBSET } I]$) and the initial routing table *InitRTable* is an empty set.

Leaf Set. The leaf set data structure *ls* of a node is modeled as a record (using syntax $[component1, component2]$) with three components: $ls.node$, $ls.left$ and $ls.right$ (a dot is used to access a component of a record in TLA$^+$). The first component contains the identifier of the node maintaining the leaf sets, the other two components are the two leaf sets to either side of the node. The following operations access or manipulate leaf sets. Here we reuse the arithmetic operations (e.g. $\geq$, $\div$) on natural numbers ($I \subseteq \mathbb{N}$).

The operation $AddToLSet(delta, ls)$ adds the set of nodes d into left and right sides of the leaf sets ls. Due to the space restriction, its complex formal definition is omitted here and can be found in [10].

Definition 1 (Operations on leaf sets ($ls \in LSet, delta \in$ SUBSET I))

$$
\begin{aligned}
LSet &\triangleq [node \in I, left \in \text{SUBSET } I, right \in \text{SUBSET } I] \\
GetLSetContent(ls) &\triangleq ls.left \cup ls.right \cup \{ls.node\} \\
EmptyLS(i) &\triangleq [node \mapsto i, left \mapsto \{\}, right \mapsto \{\}] \\
LeftNeighbor(ls) &\triangleq \text{IF } ls.left = \{\} \text{ THEN } ls.node \\
&\quad \text{ELSE CHOOSE } n \in ls.left : \forall p \in ls.left : \\
&\qquad CwDist(p, ls.node) \geq CwDist(n, ls.node) \\
RightNeighbor(ls) &\triangleq \text{IF } ls.right = \{\} \text{ THEN } ls.node \\
&\quad \text{ELSE CHOOSE } n \in ls.right : \forall q \in ls.right : \\
&\qquad CwDist(ls.node, q) \geq CwDist(ls.node, n) \\
LeftCover(ls) &\triangleq (ls.node + CwDist(LeftNeighbor(ls), ls.node) \div 2)\%2^M \\
RightCover(ls) &\triangleq (RightNeighbor(ls) + \\
&\quad CwDist(ls.node, RightNeighbor(ls)) \div 2 + 1)\%2^M \\
Covers(ls, k) &\triangleq CwDist(LeftCover(ls), k) \\
&\quad \leq CwDist(LeftCover(ls), RightCover(ls))
\end{aligned}
$$

Messages. Messages are defined as records consisting of their destinations and the message content: $DMsg \triangleq [destination \in I, mreq \in MReq]$. The message content ($MReq$) consists of different types. The actions in the dynamic model are mainly designed to handle these messages. Therefore, the different message types are formally defined here to provide better understanding of the dynamic models explained later in Section 2.2.

Definition 2 (Message Types)

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

Statuses. Together with the message types above, a brief introduction of the statuses of a node helps the understanding of the dynamic model.

$$status \in [I \to \{\text{“ready”}, \text{“ok”}, \text{“waiting”}, \text{“dead”}\}]$$

A node is initially either “ready” or “dead”. As soon as a “dead” node sends the “JoinRequest” message, it turns to the status “waiting”, which means it is waiting to become “ok”. After it has completed its leaf sets and received all the “ProbeReply” messages, it will become “ok”. Once it has obtained both leases from its left and right neighbors, it will become “ready”. Only “ready” nodes can deliver “Lookup” messages or reply to “JoinRequest” messages.

2.2 Dynamic Model

The overall system specification *Spec* is defined as $Init \wedge \Box[Next]_{vars}$, which is the standard form of TLA$^+$ system specifications. $\Box$ stands for temporal operator *always*. The whole expression requires that all runs start with a state that satisfies the initial condition *Init*, and that every transition either does not change *vars* (defined as the tuple of all state variables) or corresponds to a system transition as defined by *Next*. This form of system specification is sufficient for proving safety properties. Since liveness properties are beyond the verification interest of this paper, no fairness hypotheses are asserted, claiming that certain actions eventually occur.

Definition 3 (Overall Structure of the TLA$^+$ Specification of Pastry)

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

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

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

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

The next-state relation *Next* is a disjunction of all possible system actions, for all pairs of identifiers $i, j \in I$. Each action is defined as a TLA$^+$ action formula. Due to the page limit, we only show two formal definitions of the actions. The action $Deliver(i, k)$ (Definition 4) is referenced in the safety property and formal proof. The action $RecJReq(i)$ (Definition 5) is crucial of understanding the improvement of LuPastry in allowing only one node to handle join requests to avoid collisions caused by concurrent joins.

The action $Deliver(i, k)$ is executable if node i is "ready", if there exists an unhandled "Lookup" message addressed to i, and if j, the identifier of the requested key, falls within the coverage of node i (see Definition 1). Its effect is simply defined as removing the message m from the network, due to the fact that only the execution of the action is interesting, not the answer message that it generates. Each time it receives a message, the node will remove the message from the message pool *receivedMsgs*, so that it will not be received again. The other variables are unchanged.

Definition 4 (Action: *Deliver*(i, j))

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

The actions basically handle the different message types shown in Section 2.1. In action $Lookup(i, j)$, a node sends out a "Lookup" message, which contains only the node j it is looking for. In action $Join(i, j)$, a "JoinRequest" message is sent to node i to join a new node j. Using the same routing algorithm, the "Lookup" and "JoinRequest" messages are routed to the node which covers the key j, through several nodes via execution of $RouteJReq(i, j)$ or $RouteLookup(i, j)$ actions.

Definition 5 (Action: *RecJReq*(i))

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

In action *RecJReq*(i) (Definition 5) a “ready” node i covers the joining node in the “JoinRequest” message and has not yet started helping another node to join ($toj[i] = i$), therefore it replies to the joining node with a “JoinReply” message. It also sets *toj* to be that joining node to prevent other nodes to join through it. This is the mechanism for avoiding collision of coverage caused by concurrent join.

The “Probe” messages are handled in action *RecProbe*(i) by the receivers i. As a reply to the probing message, the node i sends a “ProbeReply” message containing the node replying to the probe (*node*), the replier’s leaf sets and a set of *failed* nodes back to the probing node. In the action *RecPReply*(i), the node i adds the sender of the “ProbeReply” message into its own leaf sets. When all awaiting probe messages have been answered, the node becomes “ok”. Consequently, it sends out “LeaseRequest” messages to update the leases of its direct left neighbor and right neighbor.

As long as a node is “ok”, it can send “LeaseRequest” messages to request leases from its direct neighbors using *RequestLease*(i). In action *RecLReq*(i), the node i replies to the lease request with a “LeaseReply” message containing its own leaf sets, where its own identifier is contained in *lset.node*. Instead of only sending back the node identifier, the leaf sets were designed to provide extra information, which, as in a “Probe” message, may serve to propagate and exchange leaf sets among nodes. If the sender is its direct neighbor, it grants the lease.

In action *RecLReply*(i), the node i updates its lease of the sender of a “LeaseReply” message, if the sender is its direct neighbor. If the node i is of status “ok” and completes leases both of its direct neighbors, then it becomes “ready”. If the node i is helping the sender to join the network, it also sets the *toj* to itself allowing it to help other nodes.

The formal model of LuPastry actions in TLA$^+$ code can be found in [10].

2.3 The Correctness Properties

Since TLA$^+$ does not have type, state variables should conform to their desired data structures, so that accessing their components will always be successful. For example, *status[i]* should access the state variable *status* of a particular node i and it is supposed to be one of the states, not a node identifier. The correctness of "types" are defined as state property *TypeInvariant* and then proved to be an invariant for the system as shown in Theorem 1.

Property 1 (*TypeInvariant*)

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

Theorem 1 (Type Correctness) $\mathit{Spec} \Rightarrow \Box \mathit{TypeInvariant}$

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

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

Property 2 (*CorrectDelivery*)

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

Observe that there can be two nodes with minimal distance from k, to either side of the key. Therefore, the asymmetry in the definition of $\mathit{LeftCover}(ls, k)$ and $\mathit{RightCover}(ls, k)$ in Definition 1 is designed to break the tie and ensure that only one node is allowed to deliver. The major verification goal is formalised in Theorem 2, that given the formulas defined for Pastry as *Spec*, it can be entailed that the property
CorrectDelivery always holds.

Theorem 2 (Correctness of Pastry) $\mathit{Spec} \Rightarrow \Box \mathit{CorrectDelivery}$

3 Theorem Proving

Model checking can only provide validation on four nodes. To get a generic verification of the Pastry protocol on arbitrary number of nodes, we need to use a theorem proving approach. Using the TLA$^+$ theorem prover TLAPS, we proved in [12] that the conjunction of *HalfNeighbor* and *NeighborClosest* implies *CorrectDelivery*. The most subtle part left is the induction proof of invariants, which extends these two properties. The proof is based on the **assumption** that there are no departure of nodes and that an active node can help at most one node to join at a time.

3.1 Inductive Proof of Invariant *HalfNeighbor*

The property *HalfNeighbor* (part of Invariant 1) is extended finally to a more complex one: *HalfNeighborExt*, stating that if there is more than one member of ReadyOK on the ring (a node is either "ready" or "ok"), then none of them will have an empty leaf set.

For the special case that there is only one member of ReadyOK k on the ring, the following statements hold:

- k has no neighbor;
- every "waiting" node (waiting to become "ok") knows at most the node k and itself;
- there is no "Probe" message to k;
- there is no "ProbeReply" message or "LeaseReply" message at all;
- the leaf set within a "JoinReply" message can only contain k.

Invariant 1 (*HalfNeighborExt*)

$$
\begin{array}{l}
\vee \forall k \in ReadyOK : RightNeighbor(lset[k]) \neq k \wedge LeftNeighbor(lset[k]) \neq k \\
\vee \exists k \in ReadyOK : \\
\quad \wedge\ ReadyOK = \{k\} \\
\quad \wedge\ LeftNeighbor(lset[k]) = k \\
\quad \wedge\ RightNeighbor(lset[k]) = k \\
\quad \wedge\ \forall \mathbf{w} \in \mathbf{NodesWait} : \mathbf{GetLSetContent}(\mathbf{lset}[\mathbf{w}]) \in \text{SUBSET } \{\mathbf{k}, \mathbf{w}\} \\
\quad \wedge\ \neg\exists \mathbf{ms} \in \mathbf{receivedMsgs} : \mathbf{ms.mreq.type} = \textit{"ProbeReply"} \\
\quad \wedge\ \neg\exists \mathbf{mk} \in \mathbf{receivedMsgs} : \wedge\ \mathbf{mk.mreq.type} = \textit{"Probe"} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \mathbf{mk.destination} \neq \mathbf{k} \\
\quad \wedge\ \forall \mathbf{mj} \in \mathbf{receivedMsgs} : \mathbf{mj.mreq.type} = \textit{"JoinReply"} \\
\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathbf{GetLSetContent}(\mathbf{mj.mreq.lset}) = \mathbf{k} \\
\quad \wedge\ \neg\exists \mathbf{mb} \in \mathbf{receivedMsgs} : \mathbf{mb.mreq.type} = \textit{"LeaseReply"}
\end{array}
$$

The formal expression shown in Invariant 1 includes the original property *HalfNeighbor* (the first 5 lines), and its extension (the remaining lines in **bold**).

This invariant is extended during the proof *HalfNeighbor* step by step. Firstly, we check what is missing as prerequisites to prove *HalfNeighbor*$'$ on its inductive proof at each action. Secondly, we strengthen *HalfNeighbor* by adding auxiliary

conjunctions in such a way that it provides exactly the prerequisite for the proof. Each time the invariant is extended, the model checker TLC is employed on the Pastry model to help check if the new invariant holds on the model of four nodes. Upon violation of such a model checking approach, the formula derived from the last state of the counterexample is used to reformulate the invariant.

3.2 Proof of *NeighborClosest*

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

Property 3 (*NeighborClosest*)

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

The intuition of searching for the appropriate invariant for proving *NeighborClosest* is backwards symbolic execution. The idea is to find a candidate invariant whose violation trace, if it is not valid, can be shorter, such that the model checker TLC can be used to help discover and improve such an invariant. Based on the assumption that no nodes leave the network and the protocol improvement in LuPastry that a "ready" node can handle at most one joining node at a time, the property *NeighborClosest* can be further reduced to the following properties: *IRN* and *NRI* (formally specified in Proerty 4).

The properties *IRN* and *NRI* together subsume the property *NeighborClosest*. The difference is that *NeighborClosest* guarantees that "ready" nodes do not ignore other "ready" nodes between themselves and their neighbors, while *IRN* and *NRI* states that every node does not ignore any "ready" nodes between itself and its neighbor.

Property 4 (*IRN* and *NRI*)

$$\begin{aligned} &IRN \triangleq \forall i \in I, r \in ReadyNodes : i \neq r \\ &\qquad \Rightarrow CwDist(i, RightNeighbor(lset[i])) \leq CwDist(i, r) \\ &NRI \triangleq \forall i \in I, r \in ReadyNodes : i \neq r \\ &\qquad \Rightarrow CwDist(LeftNeighbor(lset[i]), i) \leq CwDist(r, i) \end{aligned}$$

The properties *IRN* and *NRI* state that there cannot be a "ready" node closer to arbitrary node i, than its left and right neighbors. Since these two properties are symmetrical, we only focus on one of them in this paper, *IRN*.

Induction Invariant. Due to the page limit, we only focus on the invariant *IRN* and give intuition of the discovery of the relevant invariants used for proving *IRN*. The formal description and proof of all invariants can be found in full in [10] and are explained intuitively in [9].

Invariant 2 (*InvLuPastry*)

$$IRN \wedge TojNoReady \wedge SemToj \wedge TojClosestL \wedge GrantNeighbor \wedge GrantHistL \wedge \ldots$$

Proof Sketch of the Invariant IRN. The following proof sketch illustrates the discovery and proof of the induction invariant for proving *IRN*, the most interesting and subtle part of the formal verification approach.

Based on the definition of *IRN*, the modification of two variables *lset* and *status* is critical. Regarding the change of leaf sets *lset*, adding nodes into leaf sets preserves the validity of the invariant. Since no action in the new Pastry model removes nodes from leaf sets, the changes of leaf sets always preserve the invariant *IRN*.

Regarding the changes of *status* from "ok" to "ready" in the action *RecLReply*(r), we construct the negation of *IRN* as shown in Fig. 3: assume that a node r is turning from "ok" to "ready" in action *RecLReply*(r) and this node lies exactly between an arbitrary node i and its direct right neighbor n. The proof is to find the contradiction of this situation.

For this we need an invariant *TojNoReady*: if the leaf sets of some not yet "ready" node i is not empty, then there must exist a "ready" node, through which node i has joined the network.

$$\begin{aligned} TojNoReady \triangleq\ & \forall i \in I : i \notin \text{ReadyNodes} \wedge lset[i] \neq EmtyLS(i) \\ & \Rightarrow \exists r \in \text{ReadyNodes} : toj[r] = i \end{aligned}$$

Applying *TojNoReady* on the "ok" node r, there must be a "ready" node r_2, such that $toj[r_2] = r$. The proof method is to refute the existence of such a node r_2. According to *IRN*, node r_2 cannot be inside the range from r to its right neighbor. Hence, 3 cases are possible for the position of node r_2 as shown in Fig. 3.

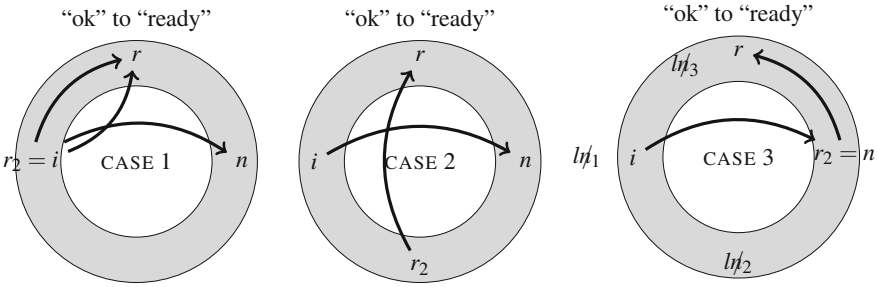

Fig. 3. Case analysis of the node r_2 w.r.t. node i and its right neighbor n.

CASE $1 : r_2 = i$. Let us introduce another invariant *SemToj* (the "semantic" of variable *toj*): if a not yet "ready" node i has joined through the "ready" node r, then node i must be r's direct neighbor. We reuse r, i as binding variables here because we can directly apply the invariant into our sub-goal.

$$\begin{aligned} SemToj \triangleq\ & \forall r, i \in I : i \notin \text{ReadyNodes} \wedge toj[r] = i \wedge r \neq i \\ & \Rightarrow RightNeighbor(lset[r]) = i \vee LeftNeighbor(lset[r]) = i \end{aligned}$$

By *SemToj* and our assumption we know that r must be a direct neighbor of r_2.
(1) If r were the right neighbor of r_2, which is now i, then r should be the right neighbor of i, which contradicts with n (see Fig. 3). (2) If r were the left neighbor of r_2, then the left distance (counterclockwise) from i to its left neighbor r is larger than the left distance from i to its right neighbor, which contradicts the definitions of *LeftNeighbor* and *RightNeighbor* in Definition 1. Hence r_2 cannot be i.
CASE 2 : $CwDist(i, RightNeighbor(lset[i])) < CwDist(i, r_2)$. Given that the node r is the direct neighbor of r_2 (shown using *SemToj*), we perform case analysis on the node r. Suppose r is the right neighbor of node r_2 as illustrated in Fig. 3 since the other case can be proved symmetrically.

To refute this possibility, let us analyze the status of node i. If node i were a "ready" node, then this would violate *IRN* for the node r. If not, we need to use invariant *TojNoReady* to construct an arbitraryly positioned "ready" node r_4, through which node i is currently joining. Then we introduce another invariant *TojClosestL*, which states that if node r is joined through some node r_2, then between these two nodes, there exists no further node such as i, which is currently joining through another node r_4. Hence, we can refute the existence of such a node r_4 and get the contradiction to close this case.

$$\begin{aligned} TojClosestL \triangleq\ & \forall r_1, r_2, i, k \in I : \\ & \wedge\ i \neq r_1 \wedge i \neq k \wedge toj[r_1] = i \wedge toj[r_2] = k \wedge r_2 \neq k \\ & \wedge\ RightNeighbor(lset[r_1]) = i \wedge i \notin \text{ReadyNodes} \\ & \Rightarrow CwDist(r_1, i) \leq CwDist(k, i) \end{aligned}$$

CASE 3 : $r_2 = n$ (n refers to $RightNeighbor(lset[i])$). Here, we make a case analysis on the position of $ln = LeftNeighbor(lset[r])$, and then close all the cases by refuting the existence of such ln.

(i) The node ln cannot be the same node as i (ln_1 in CASE 3 of Fig. 3), because according to *GrantNeighbor* (introduced below), if node ln had granted the node r, then r could not be closer than its right neighbor n.

$$\begin{aligned} GrantNeighbor \triangleq\ & \forall k, i \in I : i \neq k \wedge i \in grant[k] \\ & \Rightarrow CwDist(k, RightNeighbor(lset[k])) \leq CwDist(k, i) \wedge \ldots \end{aligned}$$

(ii) The node ln cannot be to the left of i (ln_2 in CASE 3 of Fig. 3). Since the node i cannot be "ready" due to *IRN*, let us use invariant *TojNoReady* to construct a node r_3, through which node i is currently joining. It remains to refute the existence of such a node r_3. The node r_3 cannot be r_2, hence, node r_3 must be the left neighbor of i.

Now we can do case analysis on the position of r_3 as the left neighbor of i. On the one hand, it must lie between i and ln, because if node ln is "ready", it cannot lie between a node i and its left neighbor r_3 by *NRI*. But on the other hand, node r_3 cannot lie between i and ln, because r_3 is "ready" and it should

not lie between a node r and its left neighbor ln. Therefore, r_3 can only be equal to node ln.

To force contradiction, the further invariant $GrantHistL$ is needed, which takes the facts above as precondition and derives that r_2 must be closer to i than the other node r. $GrantHistL$ states that if a not yet "ready" node i lies between two other different nodes l and r, and node i is joined through one of the nodes (e.g. l), whereas this node (i.e. l) has granted its lease to the other node (i.e. r), then the direct neighbor of i must be closer to i than the other node (i.e. r). Regarding the last case in Fig. 3, r_2 is not closer to i than r. Hence a contradiction is derived, concluding the proof of this case.

$$
\begin{aligned}
&GrantHistL \triangleq \forall l, i, r \in I : \\
&\quad toj[r] = i \wedge i \neq l \wedge l \in grant[l] \wedge i \notin \text{ReadyNodes} \wedge CwDist(l, i) < CwDist(l, r) \\
&\qquad \Rightarrow CwDist(LeftNeighbor(lset[i]), i) \leq CwDist(l, i)
\end{aligned}
$$

(iii) The node ln cannot exist between i and r (ln_3 in CASE 3 of Fig. 3). By *IRN*, ln cannot be "ready", because it lies between a node i and its right neighbor r_2. Then again by *TojNoReady*, there exists a node r_5, such that $toj[r_5] = LeftNeighbor(lset[r])$. The next step is to make a case analysis of the position of r_5. Because of *IRN*, it cannot be inside the range $[i, rn(i)]$. Because of *TojClosestL*, r_5 cannot be outside the range of (i, r). Hence, r_5 cannot exist. Hence, node ln cannot lie between i and r.

In conclusion, there is no possible position for such a node ln to exist, which means that there exists no node to grant node r its lease to make it "ready", and therefore, the constructed assumption as violation of *IRN* is impossible, completing the overall proof. □

The invariants introduced in this proof are also proved using TLAPS, and further invariants are introduced and proved. The final TLA$^+$ proof for the inductive invariant consists of more than 14,500 lines. Additionally, the type correctness is also proved inductively in about 1,000 lines. These proofs with more than 20,000 lines, corresponding to more than 10,000 proof steps, are all automatically verified using the TLAPS proof manager, which launches different back-end first-order theorem provers or an extension of ISABELLE to find the proof.

4 Conclusion, Related Work and Future Work

This paper represented a formal verification of the Pastry protocol, a fundamental building block of P2P overlay networks. To the best of my knowledge, this is the first formal verification of Pastry, although the application of formal modeling and verification techniques to P2P protocols is not entirely new. For example, 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.

Chord ([17]) is another virtual ring implementation of *DHT*. Being described with a more formal specification, it is targeted by many verification approaches, such as [14], [8], [15] and [1]. A recent approach is [18], which uses Alloy to model Chord at a high level of abstraction where operations such as *join* or *stabilize* are considered atomic and non-interfering. Focusing on eventual consistency, she found a flaw in the original description of the algorithm and suggests a repair that may be correct. However, Alloy is not supported by a theorem proving language and tools like TLAPS to formally show an understandable proof of invariants as shown in this paper.

Pastry is a reasonably complicated algorithm that mixes complex data structures, dynamic network protocols, and timed behavior for periodic node updates. LuPastry abstracts from timing aspects, which are mainly important for performance, but otherwise models the algorithm as faithfully as possible. Here a "ready" node adds the joining node as soon as it receives the join request and does not accept any new join request until it gets the confirmation that the current joining node is "ready". In fact, LuPastry has been modified iteratively until the final proof of its invariants. LuPastry is verified against the property *CorrectDelivery* through inductive proof of invariants, under the assumption that no nodes leave the network. The proof serves at the same time as evidence of correctness of the formal model with respect to the verified property *CorrectDelivery* as well as a real world example demonstrating the possibility of using TLAPS for a large scale proof consisting of more than 10,000 proof steps.

Future work will include weaker assumptions to allow some bounded departure of nodes and prove that under particular constraints, the *CorrectDelivery* can still be ensured. Future work may also include formulating liveness properties for proving availability of the system based on our validation approach; generalizing the *DHT* model based on the static model of LuPastry; and increasing the automation degree of the interactive theorem prover TLAPS based on the similar patterns I have written as part of the formal proof.

Acknowledgments. I would like to thank my PhD supervisors Christoph Weidenbach and Stephan Merz for their support on this research topic and all the thesis and paper reviewers for their valuable comments.

References

1. Bakhshi, R., Gurov, D.: Verification of peer-to-peer algorithms: A case study. Electr. Notes Theor. Comput. Sci. **181**, 35–47 (2007)
2. Borgström, J., Nestmann, U., Onana, L., Gurov, D.: Verifying a structured peer-to-peer overlay network: the static case. In: Priami, C., Quaglia, P. (eds.) GC 2004. LNCS, vol. 3267, pp. 250–265. Springer, Heidelberg (2005)
3. Castro, M., Costa, M., Rowstron, A.I.T.: Performance and dependability of structured peer-to-peer overlays. In: International Conference on Dependable Systems and Networks (DSN 2004), pp. 9–18. IEEE Computer Society, Florence (2004)
4. Haeberlen, A., Hoye, J., Mislove, A., Druschel, P.: Consistent key mapping in structured overlays. Tech. Rep. TR05-456, Rice University, Department of Computer Science, August 2005

5. Hellerstein, J.M.: Toward network data independence. ACM SIGMOD Record **32**(3), 34–40 (2003)
6. Lamport, L.: Specifying Systems, The TLA$^+$ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
7. Lamport, L.: TLA tools (2012). http://www.tlaplus.net/
8. Li, X., Misra, J., Plaxton, C.G.: Active and Concurrent Topology Maintenance. In: Guerraoui, R. (ed.) DISC 2004. LNCS, vol. 3274, pp. 320–334. Springer, Heidelberg (2004)
9. Lu, T.: Formal Verification of the Pastry Protocol. Ph.D. thesis, Universität des Saarlandes, Saarbrücken (2013). urn:nbn:de:bsz:291-scidok-55878
10. Lu, T.: The TLA$^+$ codes for the pastry model (2013). http://tiit.lu/fmPastry/
11. Lu, T., Merz, S., Weidenbach, C.: Model checking the Pastry routing protocol. In: Bendisposto, J., Leuschel, M., Roggenbach, M. (eds.) 10th Intl. Workshop Automatic Verification of Critical Systems (AVOCS), pp. 19–21. Universität Düseldorf, Düsseldorf, Germany (2010)
12. Lu, T., Merz, S., Weidenbach, C.: Towards verification of the pastry protocol using TLA$^+$. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 244–258. Springer, Heidelberg (2011)
13. Lu, T., Merz, S., Weidenbach, C.: Formal verification of the pastry protocol using TLA$^+$. 18th International Symposium on Formal Methods (2012)
14. Lynch, N., Stoica, I.: Multichord: A resilient namespace management protocol. MIT CSAIL Technical Report (2004)
15. Risson, J., Robinson, K., Moors, T.: Fault tolerant active rings for structured peer-to-peer overlays. In: The IEEE Conference on Local Computer Networks, 30th Anniversary 2005, pp. 18–25. IEEE (2005)
16. Rowstron, A., Druschel, P.: Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, p. 329. Springer, Heidelberg (2001)
17. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review **31**(4), 149–160 (2001). ACM
18. Zave, P.: Using lightweight modeling to understand chord. Computer Communication Review **42**(2), 49–57 (2012)