Towards A Certified Peer-to-Peer Lookup System

Anish Athalye
aathalye@mit.edu

Abstract

Implementing peer-to-peer systems correctly is difficult due to the extreme situations under which these systems are designed to operate. Unlike in other types of distributed systems, in peer-to-peer systems, arbitrary machines can participate in the system, and machines can leave and join the network at any time. This distinction makes it challenging to formally reason about these types of systems.

In order to better understand how to design and implement certified P2P systems supporting dynamic membership, we work towards verifying a peer-to-peer lookup system.

1 Introduction

Decentralized peer-to-peer systems can be an effective way of implementing many types of services. However, implementing correct peer-to-peer systems is challenging due to the unique properties of P2P systems. In some types of distributed systems, the system topology is static: all machines involved are known in advance, the set of machines does not change through the course of operation, and often times, all the machines are identically configured and managed by a single party. On the other hand, in P2P systems, many of these properties do not hold.

In many P2P systems, arbitrary computers can join and leave the network at any time. The machines can have very different properties in terms of hardware configuration, network connectivity, and system availability. P2P protocols are designed to handle this and work correctly while nodes join and leave the network.

Given the challenges associated with designing and implementing P2P protocols, it is desirable to have some guarantees about the correctness of the systems. Sometimes, protocol designers mathematically prove the correctness of designs on paper. Some widely-used protocols such as Kademlia [1] have also been formalized [2] using a formal specification language such as Maude [3], providing a greater degree of assurance about the correctness of the protocol. Unfortunately, approaches using modeling tools are limited due to the limited power of some of these tools, and many of the properties that are formalized in these works are fairly shallow.

As it currently stands, there is still a *formality gap* with a lack of certified *implementations* of protocols. Even if the design of a protocol is proven correct on paper, there are still a variety of bugs that could arise in the actual implementation of the protocol. End-to-end correctness proofs would provide a much stronger guarantees about the proper functioning of a system.

In the past, others have used proof assistants to prove distributed systems correct. The Verdi framework [4], a general-purpose library for reasoning about distributed systems, is the current state-of-the-art tool for the job. The framework has been used to verify simple protocols, and a partial proof of the Raft consensus protocol has been completed.

However, the Verdi framework is not designed for reasoning about peer-to-peer systems. There are several unique challenges when dealing with P2P. The set of machines participating in the system is not known in advance, and machines can join and leave at any time. In addition, desired properties of these protocols are difficult to specify and formalize.

In this paper, we describe our experience working toward a certified implementation of a peer-to-peer lookup protocol based on the Chord protocol [5]. We build this protocol as a case study hoping to gain a better understanding of how to build verified P2P systems and how to specify and prove properties about them.

2 Protocol

Our P2P lookup protocol is based on the Chord protocol [5]. Chord is a peer-to-peer lookup protocol that

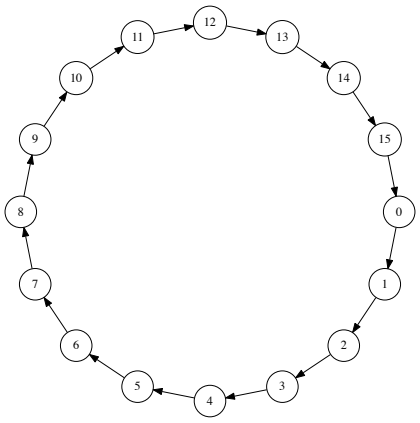

Figure 1: An example Chord ring with 16 nodes. Each node maintains only a successor pointer that points to the node with the next greatest ID.

maps 160-bit key IDs to nodes in the network that are also assigned 160-bit IDs. The protocol has been proven correct on paper, and it is used in practice for real-world applications [6, 7], so it is a good choice of protocol for a case study. Chord is highly efficient and scalable due to optimizations, but these optimizations make proofs more difficult. For the purpose of our work, we implement Chord without any optimizations such as finger tables. In our implementation, we simply have each node in the network maintain a successor pointer, as shown in Figure 1. In the worst case, messages will need to travel around the entire Chord ring to reach a node.

3 Formal model

Before we can prove theorems about the implementation of a protocol, we must have a formal system model precisely specifying the behavior of computers and the network.

3.1 Machine model

Machines in the system behave deterministically given their input. Machines maintain internal state, and machines perform computation when they receive input or network messages. Machines have two transition functions describing how their state changes when they receive local input or network messages. Upon receiving input, machines can change their internal state, produce local output, and send network messages.

A machine's behavior is summarized by its state and transition functions:

```
Inductive node_state :=
| ST_Empty : node_state
| ST_Succ : id → node_state.

Definition node_input : id → node_state → input →
    node_state × list output × list message.

Definition node_recv : id → node_state → message
    → node_state × list output × list message.
```

3.2 Network model

We considered reliable networks in our formalization of the system. In our model, the network does not duplicate or drop packets. In addition, the network preserves the order in which messages are sent – essentially, the network acts like a globally shared queue of messages. We model the network in this manner for simplicity. Support for duplicated or dropped packets would require nodes to have some form of failure detection and retransmission capability, which would complicate the protocol, and therefore complicate the proofs.

3.3 System state

In modeling overall system state, we need to track all the nodes' states and the set of messages that are in flight:

```
Record net_state := mkNetState {
  States : map id node_state;
  Messages : list message
}.
```

3.4 Step semantics

We model overall system behavior using small-step semantics, having a relation over system states. This allows us to reason about nondeterministic behavior in a clean manner. We have three inference rules – deliver, input, and join. The deliver rule (Figure 2) allows a node to process a message from the network. The input rule (Figure 3) allows a node to process a local input. The join rule (Figure 4) allows a new node to join the network initialized with an empty state.

4 Properties

There are many properties that could be desirable to have in a peer-to-peer lookup system. These properties can be divided into two categories, *safety* properties and *liveness* properties. An example of a safety property is that lookups always return the "correct" result, and an example of a liveness property is that lookups return a result in a bounded amount of time. Often times, these properties that we desire are difficult to specify precisely

```
m m_t m' : list message
m_h : message
st st' : net_state
src dst : id
body : body
q q' : node_state
n n' : map id node_state
```

$$\textsc{Deliver}\ \frac{\begin{array}{c} m = m_h : m_t \qquad st = \text{mkNetState } n\ m \\ m_h = \text{mkMessage } src\ dst\ body \\ dst \in n \qquad q = n(dst) \\ (q', m_{new}, out) = \text{node\_recv } dst\ q\ m_h \\ n' = n[dst := q'] \\ m' = m_t +\!\!+ m_{new} \qquad st' = \text{mkNetState } n'\ m' \end{array}}{st \Rightarrow st'}$$

Figure 2: Deliver inference rule.

```
m m_new m' : list message
st st' : net_state
id : id
input : input
q q' : node_state
n n' : map id node_state
```

$$\textsc{Input}\ \frac{\begin{array}{c} st = \text{mkNetState } n\ m \qquad id \in n \qquad q = n(id) \\ (q', m_{new}, out) = \text{node\_input } id\ q\ input \\ n' = n[id := q'] \\ m' = m +\!\!+ m_{new} \qquad st' = \text{mkNetState } n'\ m' \end{array}}{st \Rightarrow st'}$$

Figure 3: Input inference rule.

```
joining : id
m : list message
st st' : net_state
input : input
n n' : map id node_state
```

$$\textsc{Join}\ \frac{\begin{array}{c} joining \notin n \qquad st = \text{mkNetState } n\ m \\ n' = n[joining := ST\_Empty] \\ st' = \text{mkNetState } n'\ m \end{array}}{st \Rightarrow st'}$$

Figure 4: Join inference rule.

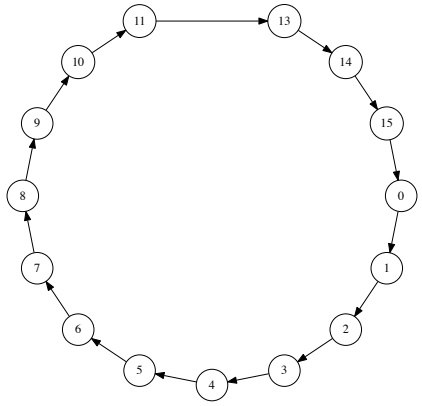

Figure 5: A quiescent network in which no nodes are attempting to join.

because the guarantees offered in P2P system are often loose guarantees. In the example of the safety property, it is difficult to formulate what is a "correct" result. We could say that a result is correct if the result of a lookup is the identifier of a node that was responsible for the key being looked up at some point between when the lookup request was sent and the response was received. Even when we can informally specify these properties, they are tricky to formalize and prove.

4.1 Towards a safety proof

As of now, we have made some progress in proving the correctness of the lookup protocol, proving a key invariant of the system. We have proven that in our system, the ring structure is always okay, in that all nodes point to the correct successor. In our joining protocol, we may have a single dangling node at a time, so we allow for this in our formalization of correct ring structure.

```
Definition ring_ok st :=
  ring_ok_quiet st ∨
  (exists joining, ring_ok_joining st joining) ∨
  (exists joining, ring_ok_attached st joining).
```

The first property, `ring_ok_quiet`, describes a quiescent network in which no nodes are pending joining (see Figure 5). The second property, `ring_ok_joining`, describes a system where a single node that is joining has sent a lookup request into the network but has not received a response (see Figure 6). The third property, `ring_ok_attached`, describes a system where a joining node has received a response and set its successor pointer.

With this description of a `ring_ok` state, we can prove that this property is an invariant of system execution – we

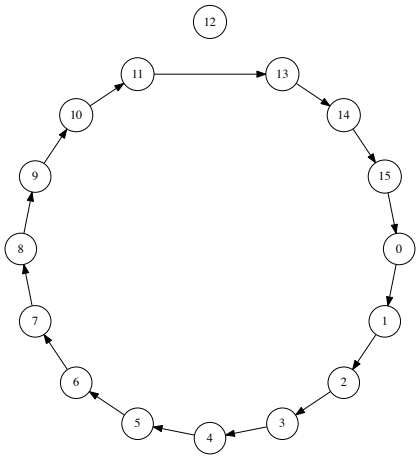

Figure 6: A system where a node is awaiting a response to a join request.

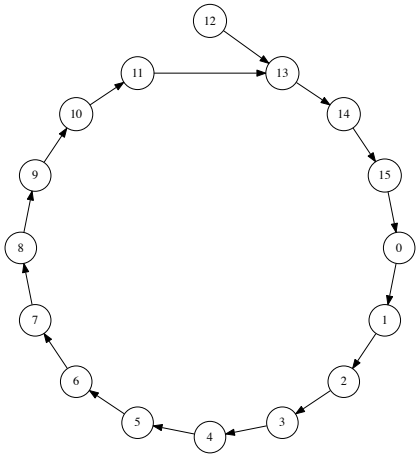

Figure 7: A system where a joining node has set its successor pointer.

Figure 8: A diagram of transitions between different types of okay states. Starting in a quiet state, the system can transition into the joining state when a node joins. While messages are being delivered, the system remains in this state. Once the node receives a response and sets its successor pointer, the system transitions into an attached state. Once the joining node informs its predecessor-to-be and it updates its pointer, the system transitions back into a quiet state.

can prove that we always stay in one of the okay states (see Figure 8).

5 Implementation

The current implementation of the protocol is approximately 150 lines of Coq code. The infrastructure and proofs make up another 850 lines of Coq code; this includes some long, manual proofs that could be automated and shortened. In addition, we have written a network simulator in Haskell so that we can interact with a simulated system and observe how it behaves. This driver code is approximately 50 lines of Haskell implementation along with about 100 lines of helper functions such as `Show` type class instances.

6 Future work

There is still much work to be done on the lookup protocol and proofs. To begin with, we intend to complete the safety proof, verifying that the system executes lookup operations correctly. After that, we want to add functionality to allow multiple nodes to leave or join the network concurrently. Then, we want to refine the network model such that we can tolerate dropped or duplicated messages and handle asynchronous message delivery. Finally, we would like to explore the possibility of adding the optimizations used in the Chord protocol, such as finger tables, and we want to be able to handle nodes crashing. Implementing and verifying functionality to handle crashes will require us to perform some sort of probabilistic reasoning in Coq; we are not aware of much work that has been done in the area.

7 Conclusion

We describe our work towards implementing a certified peer-to-peer lookup system. Our work is currently unfinished, but we have made progress in proving basic lemmas and learning how to formally reason about peer-to-peer systems. We have ideas on how to improve our work, and in the future, we hope to accomplish building high-assurance peer-to-peer software that is robust enough to be used in real-world applications.

References

[1] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, (London, UK, UK), pp. 53–65, Springer-Verlag, 2002.

[2] I. Pita and A. Riesco, "Specifying and analyzing the kademlia protocol in maude," in *9th International Workshop on Rewriting Logic and its Applications, WRLA*, 2012.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Berlin, Heidelberg: Springer-Verlag, 2007.

[4] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems." To appear in PLDI '15.

[5] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, (New York, NY, USA), pp. 149–160, ACM, 2001.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, Oct. 2007.

[7] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, Apr. 2010.