THE KADEMLIA PROTOCOL

SUCCINCTLY

BY MARC CLIFTON

SUCCINCTLY E-BOOK SERIES

Syncfusion®

The Kademlia Protocol Succinctly

By
Marc Clifton

Foreword by Daniel Jebaraj

Copyright © 2018 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.
Proofreader: John Elderkin

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Face-book to help us spread the word about the *Succinctly* series!

About the Author

Marc Clifton is a former Microsoft MVP (2004–2007) and current Code Project MVP (2005-2018). He is passionate about software architecture and all things .NET, and he also enjoys writing Python applications for the Raspberry PI and Beaglebone Single Board Computers. He loves writing (215 articles and counting on the Code Project) and exploring new technology applications (peer-to-peer networking, blockchain, big data, contextual data, declarative and functional programming, etc.). He has been a contractor for 20+ years in numerous industries, including commercial satellite design, boat yard management, emergency services record management, and insurance. When not sitting in front of the computer, Marc enjoys playing the lyre and going on adventures with his fiancee.

Chapter 1 Introduction

Kademlia, according to a paper[1] published in 2015 by Xing Shi Cai and Luc Devoyre, is "the de facto standard searching algorithm for P2P (peer-to-peer) networks on the Internet." Kademlia is a protocol specification for decentralizing peer-to-peer network operations, efficiently storing and retrieving data across the network.

A peer-to-peer network has several positive aspects:

- It is decentralized, meaning that data is not stored on a central server, but rather redundantly stored on peers.
- It is fault tolerant, meaning that if one or more peers drops out of the network, the data, having been stored on multiple peers, should still be retrievable.
- Complicated database engines are not required—the data stored on a P2P network is typically stored in key-value pairs, making it suitable for even IoT devices with limited storage to participate in the network.

Kademlia was designed by Petar Maymounkov and David Mazières in 2002. Wikipedia says this about Kademlia:

"It specifies the structure of the network and the exchange of information through node lookups. Kademlia nodes communicate among themselves using UDP. A virtual or overlay network is formed by the participant nodes. Each node is identified by a number or node ID. The node ID serves not only as identification, but the Kademlia algorithm uses the node ID to locate values (usually file hashes or keywords). In fact, the node ID provides a direct map to file hashes and that node stores information on where to obtain the file or resource."[2]

Who uses Kademlia?

Kademlia is used in file-sharing networks. For example, BitTorrent 8 uses a distributed hash table (DHT) based on an implementation of the Kademlia algorithm. The Kad network 9 uses the Kademlia protocol, with eMule being an open-source Windows client.

Why is Kademlia important?

The underlying technology of not just cryptocurrency but any blockchain, including those that implement smart contracts,[3] must include a P2P-distributed hash table, at least in regard to how blockchain technology is being discussed and applied. (Using a blockchain in a centralized scenario is sort of pointless, except perhaps for logging purposes.) Understanding how P2P

[1] http://www.tandfonline.com/doi/abs/10.1080/15427951.2015.1051674?src=recsys&journalCode=uinm20
[2] https://en.wikipedia.org/wiki/Kademlia
[3] https://en.wikipedia.org/wiki/Smart_contract

protocols work is important, as blockchain is one of those revolutionary technologies that already has, and will continue to have, an impact on software application development.

Many people think that centralized data, except for performance reasons, is on its way out.[4] As that last link states: “The more the data management industry consolidates, the more opposing forces decentralize the market.” And peer-to-peer decentralizing has built in redundancy protecting from single-point data loss and access failures. Not that decentralizing doesn’t have its own problems—security will probably be the main one, if it isn’t already.

In recognition that there are some interesting and complicated cryptocurrency and blockchain technologies coming down the road, and that these need to be understood, protocols like Kademlia are a good starting point for looking at any P2P DHT implementation. As to why Kademlia specifically, the summary to the Kademlia specification says it best:

“With its novel XOR-based metric topology, Kademlia is the first peer-to-peer system to combine provable consistency and performance, latency-minimizing routing, and a symmetric, unidirectional topology. Kademlia furthermore introduces a concurrency parameter, α, that lets people trade a constant factor in bandwidth for asynchronous lowest-latency hop selection and delay-free fault recovery. Finally, Kademlia is the first peer-to-peer system to exploit the fact that node failures are inversely related to uptime.”

Where did the name come from?

As Petar Maymounkov, one of the co-creators of Kademlia, says: “It is a Turkish word for a lucky man’ and, more importantly, is the name of a mountain peak in Bulgaria.”

Concerns with existing implementations

In perusing GitHub repositories, I found many implementations that were incomplete or clearly buggy, often discoverable simply by inspecting the code. To complicate matters further, there appear to be two different versions of the Kademlia protocol. One is a very short specification that omits much of the discussion of routing tables and performance improvements, while the longer specification on the MIT website[5] is apparently the “correct” version. The shorter specification appears to have been removed from the Rice University website.

Two implementations stand out as reasonable references:

- zencoders[6] is a C# implementation that appears to be based on the shorter specification and seems to match Jim Dixon’s description[7] of the algorithm.
- Brian Muller has an excellent implementation in Python based on the longer specification.[8]

[4] http://sandhill.com/article/is-data-decentralization-the-new-trend/

[5] https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

[6] https://github.com/zencoders/sambatyon/tree/master/Kademlia/Kademlia

[7] http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html

[8] https://github.com/bmuller/kademlia

When reviewing an open-source implementation, it is recommended that you inspect the code to verify that it implements the optimizations described in the longer specification.

Other languages

I have not looked carefully at implementations in languages other than C# and Python. There are implementations in many other languages, those being primarily written in Java, Javascript, and Go. Looking briefly at implementations in these other languages, it's fairly easy to tell which version of the specification they implement, so again, beware that depending on which specification was used, you can have very different implementations. For example, an implementation in Java makes a very specific (yet seemingly arbitrary) rule about bucket splitting (we'll get to that) that isn't found in the spec.

Requirements

The code implemented in this book requires:

- C# 7
- .NET Framework 4.7
- Visual Studio 2017

A working knowledge of LINQ is also highly recommended.

Resources used in writing this book

I used several resources in the research for this book:

- Mike De Boer's description of k-buckets
- Brian Muller's Python implementation
- zencoders' implementation
- Jim Dixon's post on the two different versions of the specification
- Jim Dixon's description of the shorter specification
- Jim Dixon's description of Section 2.4 of the specification
- Implementation of Kademlia Distributed Hash Table, Semester Thesis Written by Bruno Spori, Swiss Federal Institute of Technology Zurich
- Distributed Hash Tables and Self Organizing Networks, lecture by Jonathan Perry, MIT

Other resources to further this work

If you're interested, some additional resources to look further into P2P systems, performance, distributed ledgers, and security can be found here:

- IPFS – Content Addressed, Versions, P2P File System (Draft 3)
- S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing

- Improving Lookup Performance over a Widely-Deployed DHT
- Review on Detection and Mitigation of Sybil attack in the network, Procedia Computer Science 78 (2016) 395-401
- Flowchain: A Case Study on Building a Blockchain for the IoT
- Distributed Ledger Technology: beyond block chain, a report by the UK Government Chief Scientific Adviser

What Kademlia doesn't address

The Kademlia specification discusses how peers communicate, find each other, and share and retrieve data. There are many concerns that still need to be addressed when creating a P2P network. Here is a list of some of the major concerns (the terminology and issues will become clearer later, so consider coming back to this section once you've worked through the material):

- Key collisions in key-values: While improbable, the best way to mitigate this is to have your own peer create a random key for you.
- Peer ID collision: Again, the node ID should be created for you.
- Encrypting of values.
- Privacy of keys: While not practical with today's technology, a malicious peer could query for stored values across the entire 2^{160} key space.
- Serialization format of packets sent over the wire.
- Ability to limit what a peer stores based on value length.
- Private peer groups: Joining a public P2P network but creating a private peer group within the network.
- Partial participation: What if you want to participate in a peer network for storing and retrieving key-values, but don't want to store key-values yourself? Perhaps you're running an IoT device with limited storage?
- Registering multiple peer IDs from the same network address: This is of particular concern because you can use this to degrade the performance of a peer, as discussed in the section "Degrading a Kademlia Peer."
- A peer tampering with a value when it propagates the value to other peers: This can be remedied by including a public key to ensure the value hasn't been changed. And while it's strange to say "a peer," it becomes an issue when you download a peer application and you have no idea what's going on inside—and even if you had the source, would you know where to look?

What is accomplished here

The goal in this book is to provide a detailed discussion of an implementation of the Kademlia specification. Specifically, we will:

- Map specification with implementation
- Discover any areas of concern with the specification
- Abstract key areas of the design so that:
    - Different implementations can be selected
    - Different communication protocols can be used
    - The algorithm can be easily unit tested

Regarding unit tests

Some unit tests set up specific conditions for testing code. Others use randomly generated IDs (node IDs and keys) and verify results through a different implementation of the same algorithm. To ensure repeatability of those tests, the **`Random`** class is seeded in debug mode with the same value, and is exposed as a public static field so that some unit tests can perform their tests with a range of seeds.

Code Listing 1: Random Seeding for Unit Testing

```
#if DEBUG
 public static Random rnd = new Random(1);
 #else
 private static Random rnd = new Random();
 #endif
```

Also, most of these unit tests are really system-level tests, or at least partial-system tests. With actual unit tests of specific code sections in a particular method, well, it gets inane rather quickly. So you'll see a lot of setup stuff being done in the higher-level tests.

The unit tests presented in this book are important not just because they test the underlying implementation, but also because they demonstrate how to set up scenarios for testing the Kademlia protocol under specific conditions. Thoroughly understanding the unit tests is probably an even better way of understanding the Kademlia protocol than looking at the implementation!

A word about the code

The code presented here is incremental, meaning that as additional features are discussed and added, code is updated to reflect those new features and their unit tests. Not every refactoring is included in the code here; therefore, it is strongly recommended that if you're interested in a particular method, you review the final implementation in the actual code base.

A GitHub repository for the code base can be found here.

Referencing the Kademlia specification

This book contains numerous excerpts from the specification Kademlia: A Peer-to-peer Information System Based on the XOR Metric. These are clearly indicated in "quoted" *italicized* text.

Chapter 2 Key Concepts

The Kademlia specification consists of several sub-algorithms:

- Registering new peers
- Updating peer lists
- Obtaining the "closest peer" to a key (this concept is discussed in detail later)
- Storing and retrieving key-values
- Managing stale key-value pairs and peers

The most complex part of the code is in the registration of new peers, because this involves some magic numbers based on the Kademlia authors' research into the performance of other networks, such as Chord[9] and Pasty,[10] and the behavior of peers in those networks.

Kademlia terminology

Terms used specifically in the Kademlia specification are described here.

Overlay network: An overlay network is one in which each node keeps a (usually partial) list of other nodes participating in the network.

Node: A node (also known as a contact) is a peer in the network.

Node ID: This is a 160-bit node identifier obtained from a SHA1 hash of some key, or is randomly generated.

k-Bucket: A collection of at most k nodes (or contacts), also simply called a bucket. Each node handles up to *k* contacts within a range of IDs. Initially, the ID range is the entire spectrum from $0 <= id <= 2^{160} - 1$.

Key-Value: Peers store values based on 160-bit SHA1 hashed keys. Each stored entry consists of a key-value pair.

Router: The router manages the collection of k-buckets, and also determines into which nodes a key-value should be stored.

Distance/Closeness: The distance between a host and the key is an XOR computation of the host's ID with the key. Kademlia's most significant feature is the use of this XOR computation to determine the distance/closeness between IDs.

Prefix: A prefix is the term used to describe the *n* most significant bits (MSB) of an ID.

[9] https://en.wikipedia.org/wiki/Chord_(peer-to-peer)

[10] https://en.wikipedia.org/wiki/Pastry_(DHT)

Depth: The depth of a bucket is defined as the shared prefix of a bucket. Because buckets are associated with ranges from 2^i to $2^{i+1} - 1$ where $0 <= i < 160$, one could say that the depth of a bucket is 160 - i. We'll see later that this may not be the case.

Bucket Split: A bucket split potentially happens when a node's k-bucket is full—meaning it has *k* contacts—and a new contact with a given 160-bit key wants to register within the bucket's range for that key. At this point, an algorithm kicks in that:

- Under one condition, splits the bucket at the range midpoint into two ranges, placing contacts into the appropriate new buckets.
- Under a second condition, splits the bucket when a specific depth qualifier is met.
- Under a third condition, replaces a peer that no longer responds with the newest contact that is in a "pending" queue for that bucket.

Communication protocol

The Kademlia protocol consists of four remote procedure calls (RPCs). All RPCs require that the sender provides a random RPC ID, which must be echoed by the recipient: *"In all RPCs, the recipient must echo a 160-bit random RPC ID, which provides some resistance to address forgery; PINGS can also be piggy-backed on RPC replies for the RPC recipient to obtain additional assurance of the sender's network address."*

Anytime a peer is contacted with any of the four RPCs, it goes through the process of adding or updating the contact in its own list. The concept of "closeness" will be discussed in detail later.

Ping

"The PING RPC probes a node to see if it is online." This is considered a "primitive" function, in that it just returns the random RPC ID that accompanied the **`Ping`** request.

Store

`STORE` instructs a node to store a (key,value) pair for later retrieval. This is also considered a "primitive" function, as it again just returns the random RPC ID that accompanied the **`STORE`** request*. "To store a (key,value) pair, a participant locates the k closest nodes to the key and sends them STORE RPCS."* The participant does this by inspecting its own k-closest nodes to the key.

FindNode

"FIND_NODE takes a 160-bit ID as an argument. The recipient of the RPC returns (IP address, UDP port, Node ID) triples for the k nodes it knows about closest to the target ID. These triples can come from a single k-bucket, or they may come from multiple k-buckets if the closest k-bucket is not full. In any case, the RPC recipient must return k items (unless there are fewer than k nodes in all its k-buckets combined, in which case it returns every node it knows about)."

In an abstracted communication protocol, the recipient needs to return information about the protocol: the kind of protocol and whatever is required to contact a peer using that protocol. If multiple protocols are supported, we can consider two options:

- Return node information only for the protocols that the requester says it supports.
- Alternatively (and not as good an option), the requester can filter out returned nodes whose protocols aren't supported.

Other considerations when supporting multiple protocols are:

- The peer itself may support multiple protocols, so it should probably indicate what those are when it registers with another peer.
- The peer may have a preferred protocol.

None of the issues of different protocols is discussed in the spec—this is purely my own enhancement.

The **`FindNode`** protocol has two purposes:

- A peer can issue this RPC on contacts it knows about, updating its own list of "close" peers.
- A peer may issue this RPC to discover other peers on the network.

FindValue

"FIND_VALUE behaves like FIND_NODE—returning (IP address, UDP port, Node ID) triples—with one exception. If the RPC recipient has received a STORE RPC for the key, it just returns the stored value."

If the **`FindValue`** RPC returns a list of other peers, it is up to the requester to continue searching for the desired value from that list. Also, note this technique for caching key-values:

"To find a (key,value) pair, a node starts by performing a lookup to find the k nodes with IDs closest to the key. However, value lookups use FIND_VALUE rather than FIND_NODE RPCS. Moreover, the procedure halts immediately when any node returns the value. For caching purposes, once a lookup succeeds, the requesting node stores the (key,value) pair at the closest node it observed to the key that did not return the value."

Other considerations

Expiration time

"Additionally, each node republishes (key,value) pairs as necessary to keep them alive, as described later in Section 2.5. This ensures persistence (as we show in our proof sketch) of the (key,value) pair with very high probability. For Kademlia's current application (file sharing), we also require the original publisher of a (key,value) pair to republish it every 24 hours. Otherwise, (key,value) pairs expire 24 hours after publication, so as to limit stale index information in the system. For other applications, such as digital certificates or cryptographic hash to value mappings, longer expiration times may be appropriate."

If we want to consider using Kademlia in a distributed ledger implementation, it seems necessary that key-values never expire—otherwise, this would result in an integrity loss of the ledger data.

Over-caching

"Because of the unidirectional nature of the topology, future searches for the same key are likely to hit cached entries before querying the closest node. During times of high popularity for a certain key, the system might end up caching it at many nodes. To avoid "over-caching," we make the expiration time of a (key,value) pair in any node's database exponentially inversely proportional to the number of nodes between the current node and the node whose ID is closest to the key ID. While simple LRU eviction would result in a similar lifetime distribution, there is no natural way of choosing the cache size, since nodes have no a priori knowledge of how many values the system will store."

Bucket refreshes

"Buckets are generally kept fresh by the traffic of requests traveling through nodes. To handle pathological cases in which there are no lookups for a particular ID range, each node refreshes any bucket to which it has not performed a node lookup in the past hour. Refreshing means picking a random ID in the bucket's range and performing a node search for that ID."

Joining a network

"To join the network, a node u must have a contact to an already participating node w. u inserts w into the appropriate k-bucket. u then performs a node lookup for its own node ID. Finally, u refreshes all k-buckets further away than its closest neighbor. During the refreshes, u both populates its own k-buckets and inserts itself into other nodes' k-buckets as necessary."

Chapter 3 Getting Started

First, let's cover some basic implementation requirements for a node.

The BigInteger class

We could write our own byte array manipulation and comparison operators, which is what zencoders did, or we could use the **`BigInteger`** class in the System.Numerics namespace to handle the range of IDs from 0 <= id <= 2^{160} - 1. This is much more convenient than implementing all the bitwise logic and unit tests! As a side node, I was impressed with Python's ability to handle these values without any special classes:

Code Listing 2: Python Big Numbers

```
>>> 2 ** 160

1461501637330902918203684832716283019655932542976L
```

The node specification

"Participating computers each have a node ID in the 160-bit key space," which is simple enough. However, beware of how buckets in a node are initialized. In the shorter specification, it is implied that a node has 160 k-buckets, one for each bit in the 160-bit SHA1 hash. The longer specification discusses bucket splitting and "*initially, a node u's routing tree has a single node—one k-bucket covering the entire ID space.*" By looking at an implementation's initialization of the k-buckets, one can immediately determine which specification the code is written against.

A framework for the implementation

The initial framework for the implementation consists of the following classes:

- **`Dht`**: The peer's entry point for interacting with other peers.
- **`Router`**: Manages peer (node) lookups for acquiring nearest peers and finding key-value pairs.
- **`Node`**: Provides the **`Ping`**, **`Store`**, **`FindNode`**, and **`FindValue`** implementations.
- **`BucketList`**: Manages the contacts (peers) in each bucket and the algorithm for adding contacts (peers) to a particular bucket.
- **`ID`**: Implements a wrapper for **`BigInteger`** and various helper methods and operator overloads for the XOR logic
- **`Contact`**: Maintains the protocol that the contact (peer) uses, its ID, and **`LastSeen DateTime`**, which is used for determining whether a peer should be tested for eviction.

- **`KBucket`**: Retains the collection of contacts (peers) that are associated with a specific bucket, implements the bucket splitting algorithm, and provides other useful methods for obtaining information regarding the bucket.

These interact as illustrated in the model diagram:

- Blue: classes
- Orange: interfaces
- Purple: collections
- Green: value type fields

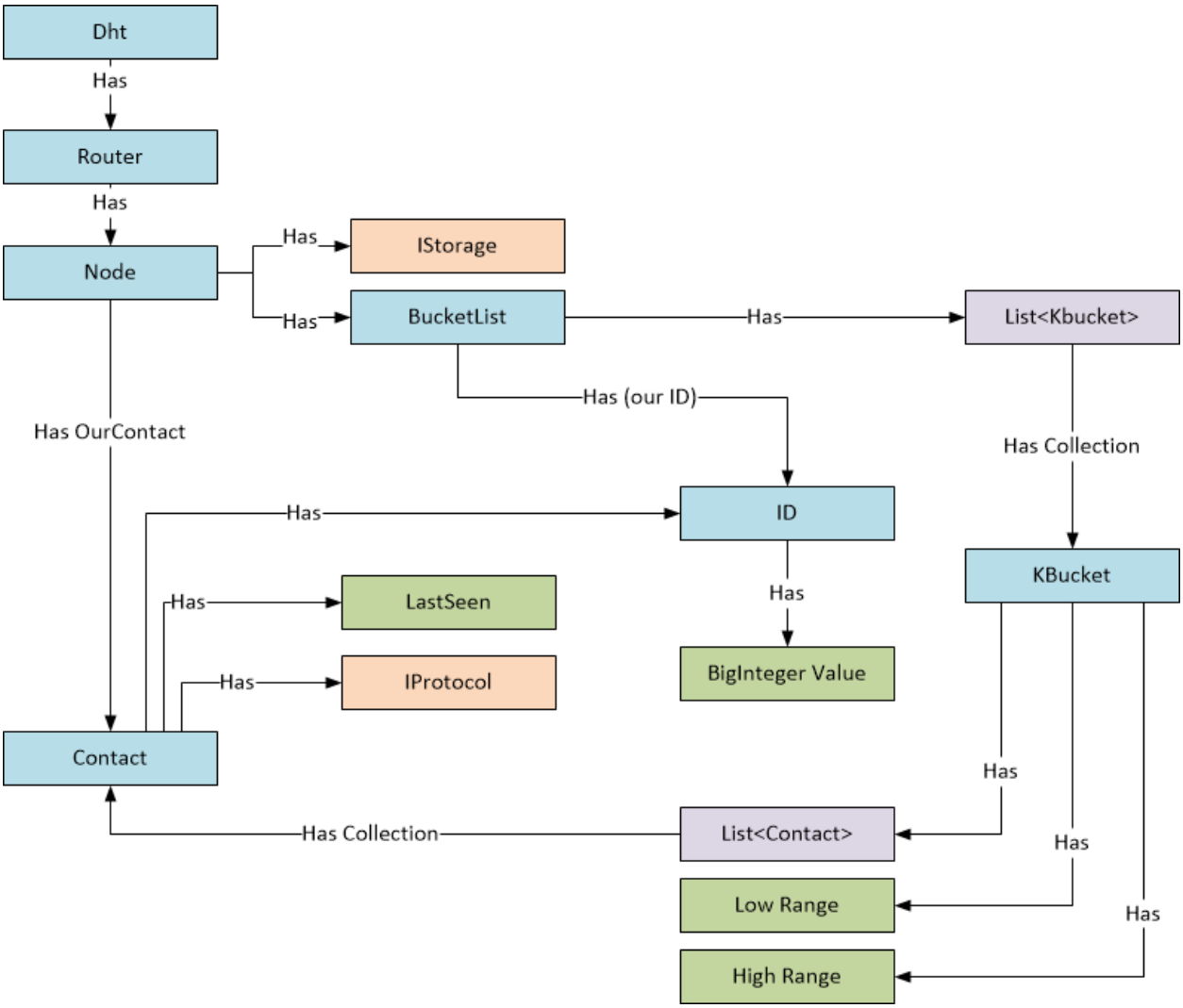

Figure 1*: The Class Model*

The ID class: initial implementation

Code Listing 3: ID Class

```
public class ID
{
  public BigInteger Value { get { return id; } set { id = value; } }

  protected BigInteger id;

  public ID(byte[] data)
  {
    IDInit(data);
  }

  public ID(BigInteger bi)
  {
    id = bi;
  }


  protected void IDInit(byte[] data)
  {
    Validate.IsTrue<IDLengthException>(data.Length ==
Constants.ID_LENGTH_BYTES, "ID
       must be " + Constants.ID_LENGTH_BYTES + " bytes in length.");
    id = new BigInteger(data.Append0());
}
```

Note: The `Validate` class defines some helper methods for asserting the condition and throwing the exception specified as the generic parameter.

There are two things to note here. First, we append the byte array with a **`0`** to force unsigned values in the **`BigInteger`**. If we don't do this, any byte array where the MSB of **`byte[0]`** is set will be treated as a negative number, which we don't want when we are comparing the range of a bucket. This is handled by a simple extension method, as in Code Listing 4.

Code Listing 4: The Append0 Extension Method

```
public static byte[] Append0(this byte[] b)
{
  return b.Concat(new byte[] { 0 }).ToArray();
}
```

Second, the byte array is in little-endian order, meaning that the least significant byte is stored first. This can get confusing when contrasted with the way that the Kademlia specification talks about ID “prefixes” and the way we tend to represent bits in an array. For example, this ID, in bits, is written out in big-endian order:

11001000 00001100

The 6-bit prefix would be **110010**. As a **BigInteger**, the array is in little-endian order, so the bytes are ordered such that the least significant byte (the LSB) is stored first:

00001100 11001000

For convenience, we implement a couple of helper properties. An extension method **Bits** is used to create a fluent method-chaining coding style.

Code Listing 5: Getting the bits in Big Endian Order

```
public string AsBigEndianString
{
  get
  {
    return String.Join(“”, Bytes.Bits().Reverse().Select(b => b ? "1" :
"0"));
  }
}


public bool[] AsBigEndianBool
{
  get
  {
    return Bytes.Bits().Reverse().ToArray();
  }
}
```

Also notice the local method, which is a C# 7 feature.

Code Listing 6: Bits Extension Method

```
public static IEnumerable<bool> Bits(this byte[] bytes)
{
  IEnumerable<bool> GetBits(byte b)
  {
    byte shifter = 0x01;

    for (int i = 0; i < 8; i++)
    {
      yield return (b & shifter) != 0;
      shifter <<= 1;
    }
  }

  return bytes.SelectMany(GetBits);
}
```

The ID class: unit tests

We can implement a few simple unit tests to verify the implementation:

Code Listing 7: ID Unit Tests

```
[TestClass]
public class IDTests
{
  [TestMethod]
  public void LittleEndianTest()
  {
    byte[] test = new byte[20];
```

```
    test[0] = 1;
    Assert.IsTrue(new ID(test) == new BigInteger(1), "Expected value to
be 1.");
  }

  [TestMethod]
  public void PositiveValueTest()
  {
    byte[] test = new byte[20];
    test[19] = 0x80;
    Assert.IsTrue(new ID(test) == BigInteger.Pow(new BigInteger(2), 159),
"Expected
     value to be 1.");
  }

  [TestMethod, ExpectedException(typeof(IDLengthException))]
  public void BadIDTest()
  {
    byte[] test = new byte[21];
    new ID(test);
  }

  [TestMethod]
  public void BigEndianTest()
  {
    byte[] test = new byte[20];
    test[19] = 0x80;
    Assert.IsTrue(new ID(test).AsBigEndianBool[0] == true, "Expected big
endian bit
       15 to be set.");
```

```
    Assert.IsTrue(new ID(test).AsBigEndianBool[8] == false, "Expected big
endian bit
       7 to NOT be set.");
  }
}
```

The Router class

At the moment, the **`Router`** class simply manages the host's node (the host is our peer).

Code Listing 8: Stub for the Router

```
public class Router
{
  protected Node node;

  public Router(Node node)
  {
    this.node = node;
  }
```

The Contact class

The **`Contact`** class manages the contact's **`ID`**, **`LastSeen`**, and network connectivity. Because I want to abstract the way network protocols are handled such that it is easy to test nodes in a virtual (in-memory) network, or nodes that use different protocols (UDP, TCP/IP, WebSockets, etc.), the network protocol is abstracted in an interface.

Code Listing 9: Basic Contact Class

```
Public class Contact  : Icomparable
{
  public IProtocol Protocol { get; set; }
  public DateTime LastSeen { get; set; }
  public ID ID { get; set; }
```

```
  /// <summary>
  /// Initialize a contact with its protocol and ID.
  /// </summary>
  public Contact(IProtocol protocol, ID contactID)
  {
    Protocol = protocol;
    ID = contactID;
    Touch();
  }

  /// <summary>
  /// Update the fact that we've just seen this contact.
  /// </summary>
  public void Touch()
  {
    LastSeen = DateTime.Now;
  }
```

The KBucket class

Each k-bucket maintains a list of up to *k* contacts.

Code Listing 10: The KBucket Class

```
public class KBucket
{
  public DateTime TimeStamp { get; set; }
  public List<Contact> Contacts { get { return contacts; } }
  public BigInteger Low { get { return low; } set { low = value; } }
  public BigInteger High { get { return high; } set { high = value; } }
```

```
  public bool IsBucketFull { get { return contacts.Count == Constants.K;
} }


  protected List<Contact> contacts;
  protected BigInteger low;
  protected BigInteger high;

  public KBucket()
  {
    contacts = new List<Contact>();
    low = 0;
    high = BigInteger.Pow(new BigInteger(2), 160);
  }


/// <summary>
/// Initializes a k-bucket with a specific ID range.
/// </summary>
  public KBucket(BigInteger low, BigInteger high)
  {
    contacts = new List<Contact>();
    this.low = low;
    this.high = high;
  }


  public void Touch()
  {
    TimeStamp = DateTime.Now;
```

```
  }

  public void AddContact(Contact contact)
  {
    Validate.IsTrue<TooManyContactsException>(contacts.Count <
Constants.K, "Bucket
       is full");
    contacts.Add(contact);
  }
}
```

The KBucket class: a unit test

A simple start for a unit test for the **`KBucket`** class is simply to verify that an exception is thrown when too many contacts are added.

Code Listing 11: Basic KBucket Unit Test

```
[TestClass]
public class KBucketTests
{
  [TestMethod, ExpectedException(typeof(TooManyContactsException))]
  public void TooManyContactsTest()
  {
    KBucket kbucket = new KBucket();

    // Add max # of contacts.
    Constants.K.ForEach(n => kbucket.AddContact(new Contact(null, new
ID(n))));

    // Add one more.
    kbucket.AddContact(new Contact(null, new ID(21)));
  }
```

```
}
```

Notice the extension method **`ForEach`**.

Code Listing 12: ForEach Extension Method

```
public static void ForEach(this int n, Action<int> action)
{
  for (int i = 0; i < n; i++)
  {
    action(i);
  }
}
```

Again, this is used for a more fluent method chaining syntax.

The BucketList class

The **`BucketList`** class is a high-level singleton container for buckets and operations that manipulate buckets. For the moment, most of this is stubbed with minimal behavior.

Code Listing 13: The BucketList Class

```
public class BucketList : IBucketList
{
  public List<KBucket> Buckets { get { return buckets; } }

  public ID OurID { get { return ourID; } }

  protected List<KBucket> buckets;

  public BucketList(ID id)
  {
    ourID = id;
```

```
    buckets = new List<KBucket>();


   // First kbucket has max range.
    buckets.Add(new KBucket());
  }

  public void AddContact(Contact contact)
  {
   // To be implemented…
  }
}
```

The Node class

The **Node** class is another high-level singleton container for handling the Kademlia commands sent over the wire. This is mostly stubbed for now.

Code Listing 14: The Node Class

```
public class Node : INode
{
  public Contact OurContact { get { return ourContact; } }
  public IBucketList BucketList { get { return bucketList; } }
  public IStorage Storage { get { return storage; } set { storage =
value; } }

  protected Contact ourContact;
  protected IBucketList bucketList;
  protected IStorage storage;

  public Node(Contact contact, IStorage storage, IStorage cacheStorage =
null)
  {
    ourContact = contact;
```

```
    bucketList = new BucketList(contact.ID);
    this.storage = storage;
  }

  public Contact Ping(Contact sender)
  {
     // To be implemented…

    return ourContact;
  }

  public void Store(Contact sender, ID key, string val)
  {
    // To be implemented…
  }

  public (List<Contact> contacts, string val) FindNode(Contact sender, ID
key)
  {
    // To be implemented…
  }

  public (List<Contact> contacts, string val) FindValue(Contact sender,
ID key)
  {
    // To be implemented…
  }
}
```

Of note here is the interface **IStorage**, which abstracts the storage mechanism for key-value pairs. Ultimately, **IStorage** will implement the following methods from Code Listing 15.

Code Listing 15: The IStorage Interface

```
public interface IStorage
{
  bool Contains(ID key);
  bool TryGetValue(ID key, out string val);
  string Get(ID key);
```

```
  string Get(BigInteger key);
  DateTime GetTimeStamp(BigInteger key);
  void Set(ID key, string value, int expirationTimeSec = 0);
  int GetExpirationTimeSec(BigInteger key);
  void Remove(BigInteger key);
  List<BigInteger> Keys { get; }
  void Touch(BigInteger key);
}
```

The Dht class

The **Dht** class is the "server"—the entry point for instantiating our peer. At the moment, the **Dht** class is simply a container for the **Router**:

Code Listing 16: Stub for the Dht Class

```
public class Dht : IDht
{
  public BaseRouter Router { get { return router; } }
  protected BaseRouter router;
}
```

Chapter 4 Adding Contacts

Version 2, Section 2.2 of the specification initially states this simple algorithm for dealing adding contacts:

"*When a Kademlia node receives any message (request or reply) from another node, it updates the appropriate k-bucket for the sender's node ID. If the sending node already exists in the recipient's k-bucket, the recipient moves it to the tail of the list. If the node is not already in the appropriate k-bucket and the bucket has fewer than k entries, then the recipient just inserts the new sender at the tail of the list. If the appropriate k-bucket is full, however, then the recipient pings the k-bucket's least-recently seen node to decide what to do. If the least recently seen node fails to respond, it is evicted from the k-bucket and the new sender inserted at the tail. Otherwise, if the least-recently seen node responds, it is moved to the tail of the list, and the new sender's contact is discarded.*"

Let's define a few terms for clarity:

- *Head of the list*: The first entry in the list.
- *Tail of the list*: The last entry in the list.
- *The appropriate k-bucket for the sender's node ID*: The k-bucket for which the sender's node ID is in the range of the k-bucket.

Figure 2 is a flowchart of what the spec says.

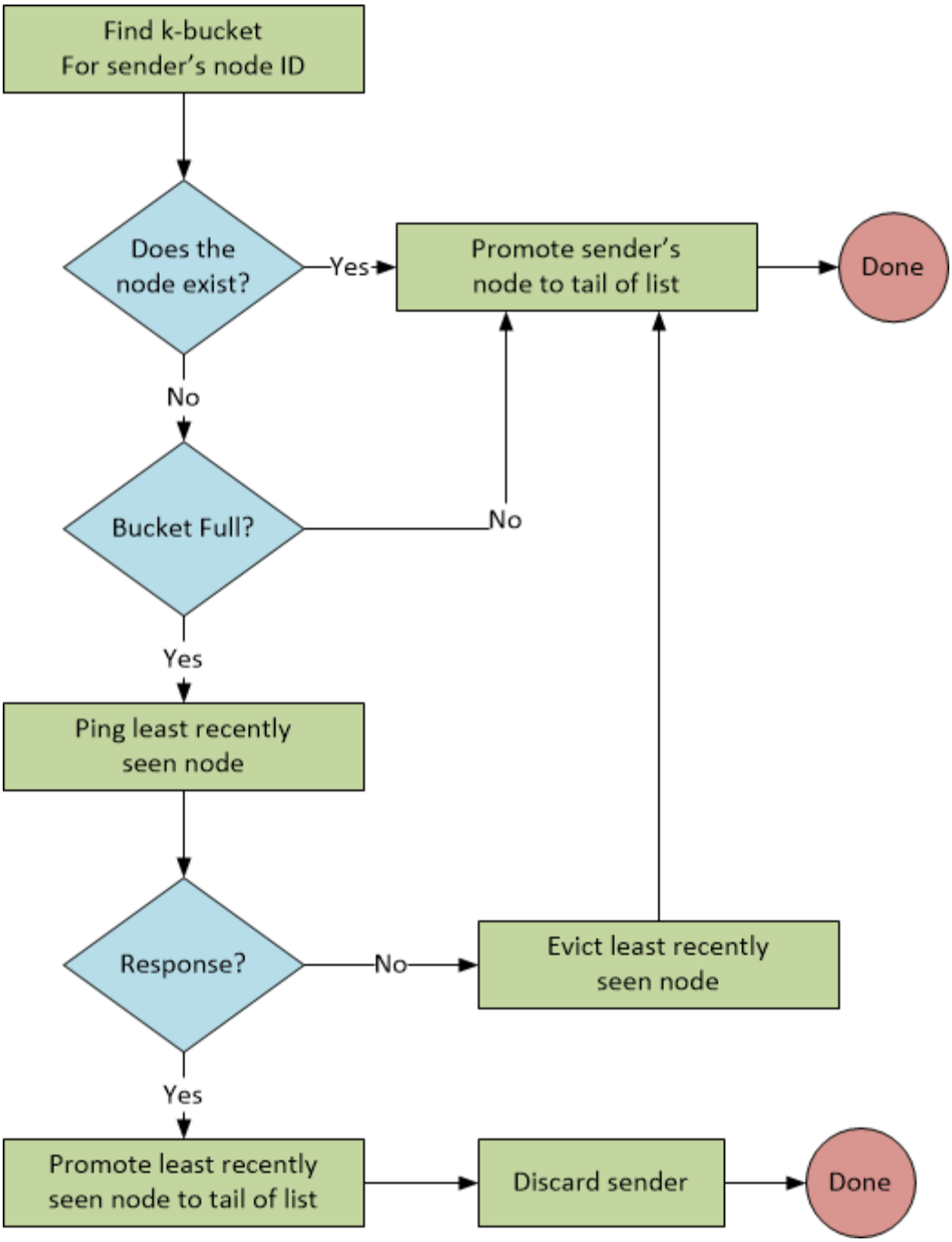

Figure 2: The Add Contact Algorithm

This seems reasonable, and the spec goes on to state:

"*k-buckets effectively implement a least-recently seen eviction policy, except that live nodes are never removed from the list. This preference for old contacts is driven by our analysis of Gnutella trace data collected by Saroiu, et. al. ... The longer a node has been up, the more likely it is to remain up another hour. By keeping the oldest live contacts around, k-buckets maximize the probability that the nodes they contain will remain online. A second benefit of k-buckets is that they provide resistance to certain DoS attacks. One cannot flush the nodes' routing state by flooding the system with new nodes. Kademlia nodes will only insert the new nodes in the k-buckets when old nodes leave the system.*"

We also observe that this has nothing to do with binary trees, which is something version 2 of the spec introduced. This is basically a hangover from version 1 of the spec. However, Section 2.4 states something slightly different:

"*Nodes in the routing tree are allocated dynamically, as needed. Initially, a node u's routing tree has a single node—one k-bucket covering the entire ID space. When u learns of a new contact, it attempts to insert the contact in the appropriate k-bucket. If that bucket is not full, the new contact is simply inserted. Otherwise, if the k-bucket's range includes u's own node ID, then the bucket is split into two new buckets, the old contents divided between the two, and the insertion attempt repeated. If a k-bucket with a different range is full, the new contact is simply dropped.*"

Again, a definition of terms is useful:

- *u's routing tree*: The host's bucket list.
- *If a k-bucket with a different range is full*: Meaning, a k-bucket that does not include u's own node ID.

Bucket-splitting

The purpose of allowing a bucket to split if it contains the host's node ID is so that the host keeps a list of nodes that are "close to it"—closeness is defined essentially by the integer difference of the node IDs, not the XOR difference (more on this whole XOR thing later).

So, this algorithm looks like Figure 3.

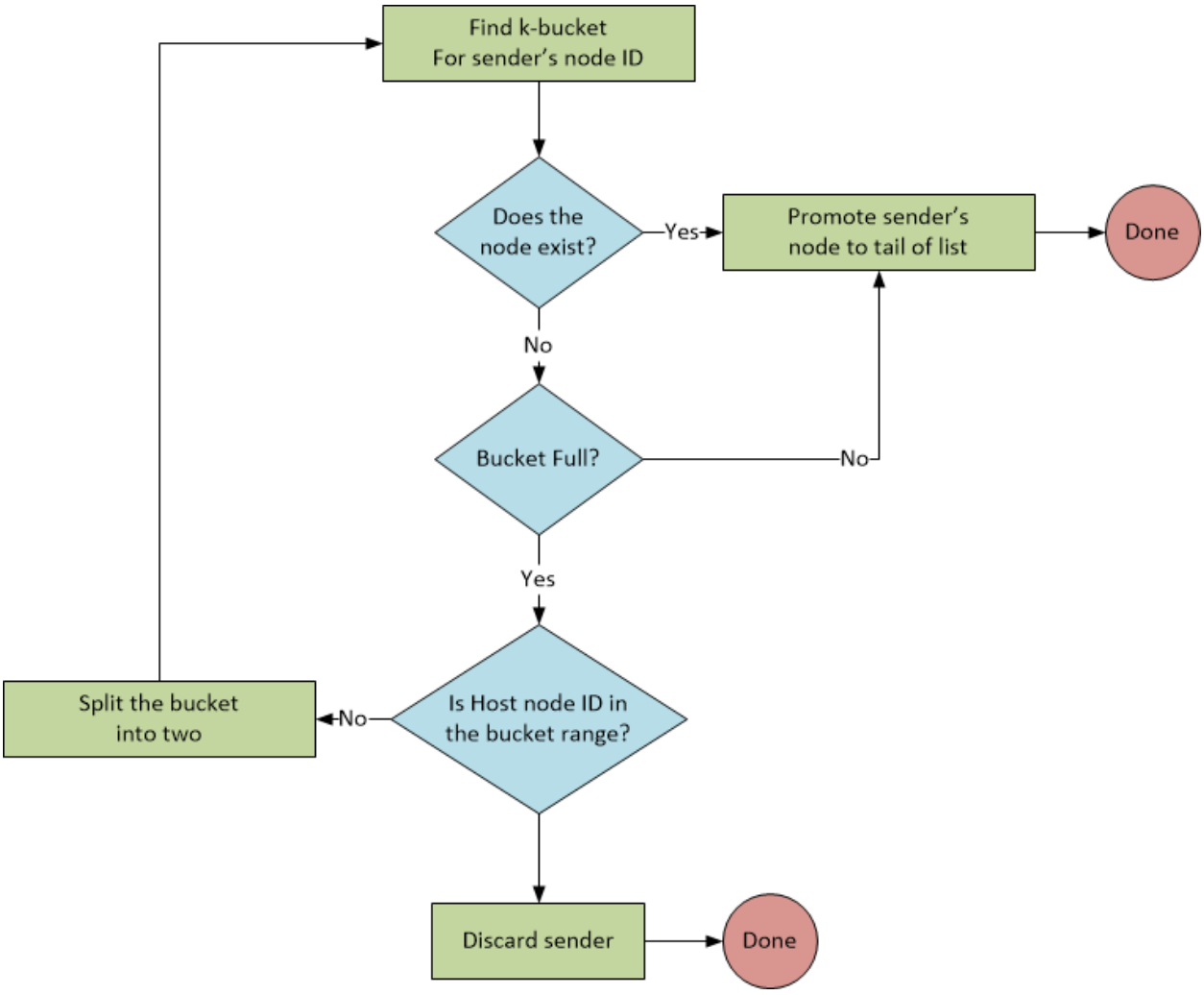

Figure 3: Bucket Splitting

What happened to pinging the least-seen contact and replacing it? Again, the spec then goes on to say:

"*One complication arises in highly unbalanced trees. Suppose node u joins the system and is the only node whose ID begins 000. Suppose further that the system already has more than k nodes with prefix 001. Every node with prefix 001 would have an empty k-bucket into which u should be inserted, yet u's bucket refresh would only notify k of the nodes. To avoid this problem, Kademlia nodes keep all valid contacts in a subtree of size at least k nodes, even if this requires splitting buckets in which the node's own ID does not reside. Figure 5 illustrates these additional splits.*"

A few more clarifications:

- *A subtree of size at least k nodes*: The reason the subtree contains "at least k nodes" is that when the parent is split, it creates two subtrees whose total number of nodes begins with k but may contain more than k nodes as nodes are added to each branch (or the branch splits again into two more branches.)

- *Even if this requires splitting buckets in which the node's own ID does not reside*: Not only is this contradictory, but there's no explanation of what "even if this requires" means. How do you code this?

This section of the specification apparently creates much confusion—I found several links with people asking about this section. It's unfortunate that the original authors themselves do not answer these questions. Jim Dixon has a very interesting response on The Mail Archive ,[11] which I present in full here:

> "The source of confusion is that the 13-page version of the Kademlia uses the same term to refer to two different data structures. The first is well-defined: k-bucket i contains zero to k contacts whose XOR distance is [2^i..2^(i+1)). It cannot be split. The current node can only be in bucket zero, if it is present at all. In fact, its presence would be pointless or worse.
>
> The second thing referred to as a k-bucket doesn't have the same properties. Specifically, the current node must be present, it wanders from one k-bucket to another, these k-buckets can be split, and there are sometimes ill-defined constraints on the characteristics of subtrees of k-buckets, such as the requirement that "Kademlia nodes keep all valid contacts in a subtree of size of at least k nodes, even if this requires splitting buckets in which the node's own ID does not reside" (section 2.4, near the end).
>
> In a generous spirit, you might say that the logical content of the two descriptions is the same. However, for someone trying to implement Kademlia, the confusion of terms causes headaches—and leads to a situation where all sorts of things are described as Kademlia, because they can be said to be, if you are of a generous disposition. However, not surprisingly, they don't interoperate."

So, my decision, given the ambiguity of the spec, is to ignore this, because as you will see next, there is yet another version of how contacts are added.

In Section 4.2, on Accelerated Lookups, we have a different specification for how contacts are added:

"*Section 2.4 describes how a Kademlia node splits a k-bucket when the bucket is full and its range includes the node's own ID. The implementation, however, also splits ranges not containing the node's ID, up to b - 1 levels. If b = 2, for instance, the half of the ID space not containing the node's ID gets split once (into two ranges); if b = 3, it gets split at two levels into a maximum of four ranges, etc. The general splitting rule is that a node splits a full k-bucket if the bucket's range contains the node's own ID or the depth d of the k-bucket in the routing tree satisfies d (mod b) != 0.*"

Another term to consider:

Depth. According to the spec: *"The depth is just the length of the prefix shared by all nodes in the k-bucket's range."* Do not confuse that with this statement in the spec: *"Define the depth, h, of a node to be 160 - i, where i is the smallest index of a nonempty bucket."* The former is referring to the depth of a k-bucket, the latter the depth of the node.

[11] https://www.mail-archive.com/p2p-hackers@lists.zooko.com/msg00042.html

With regard to the definition of *depth*, does this mean “the length of prefix shared by any node that would reside in the k-bucket’s range,” or does it mean “the length of a prefix shared by all nodes currently in the k-bucket”?

If we look at Brian Muller’s implementation in Python, we see the latter case.

Code Listing 17: Depth is Determined by the Shared Prefixes

```
def depth(self):

  sp = sharedPrefix([bytesToBitString(n.id) for n in
self.nodes.values()])

  return len(sp)


def sharedPrefix(args):

  i = 0

  while i < min(map(len, args)):

   if len(set(map(operator.itemgetter(i), args))) != 1:

     break

    i += 1

  return args[0][:i]
```

Here, the depth is determined by the shared prefixes in the nodes. So, when we use the following algorithm to determine whether a bucket can be split.

Code Listing 18: Bucket Split Logic

```
kbucket.HasInRange(ourID) || ((kbucket.Depth() % Constants.B) != 0)
```

What is this actually doing?

- First, the **`HasInRange`** is testing whether our node ID is close to the contact’s node ID. If our node ID is in the range of the bucket associated with the contact’s node ID, then we know the two nodes are “close” in terms of the integer difference. Initially, the range spans the entire 2^{160} ID space, so everybody is “close.” This test of closeness is refined as new contacts are added.
- Regarding the depth mod 5 computation, I asked Brian Muller this question: “Is the purpose of the depth to limit the number of ‘new’ nodes that a host will maintain (ignoring for the moment the issue of pinging an old contact to see if can be replaced)?” He replied: “Yes! The idea is that a node should know about nodes spread across the network—though definitely not all of them. The depth is used as a way to control how ‘deep’ a node’s understanding of the network is (and the number of nodes it knows about).”

The depth to which the bucket has split is based on the number of bits shared in the prefix of the contacts in the bucket. With random IDs, this number will initially be small, but as bucket ranges become more narrow from subsequent splits, more contacts will begin the share the same prefix and the bucket when split, will result in less “room” for new contacts. Eventually, when the bucket range becomes narrow enough, the number of bits shared in the prefix of the contacts in the bucket reaches the threshold *b*, which the spec says should be 5.

Failing to add yourself to a peer: self-correcting

Let’s say you’re a new node and you want to register yourself with a known peer, but that peer is maxed out for the number of contacts that it can hold in the particular bucket for your ID—the depth is *b*. In this case, you will not be added to the peer’s contact list. The Kademlia spec indicates that peers that fail to be added to a bucket be placed into a queue of pending contacts, which we discuss later. Regardless, not being added to a peer’s bucket list is not really an issue. Whether you are successfully added as a contact or not, you will receive back “nearby” peers. Any time you contact those peers (to store a value, for example), all the peers that you contact will try to add your contact. When one succeeds, your contact ID will be disseminated slowly through the network.

Degrading a Kademlia peer

Given a peer with both an ID and a single contact ID of less than 2^{159}, it will initially split once for 20 contacts that are added having IDs greater than 2^{159}. All of those 20 contacts having IDs greater than 2^{159} will go in the second bucket. The peer ID will not be in this second bucket range. If those contacts in the second bucket have IDs where the number of shared bits is **b**, and therefore **`b mod b == 0`**, any new contact in the range of the second bucket will not be added.

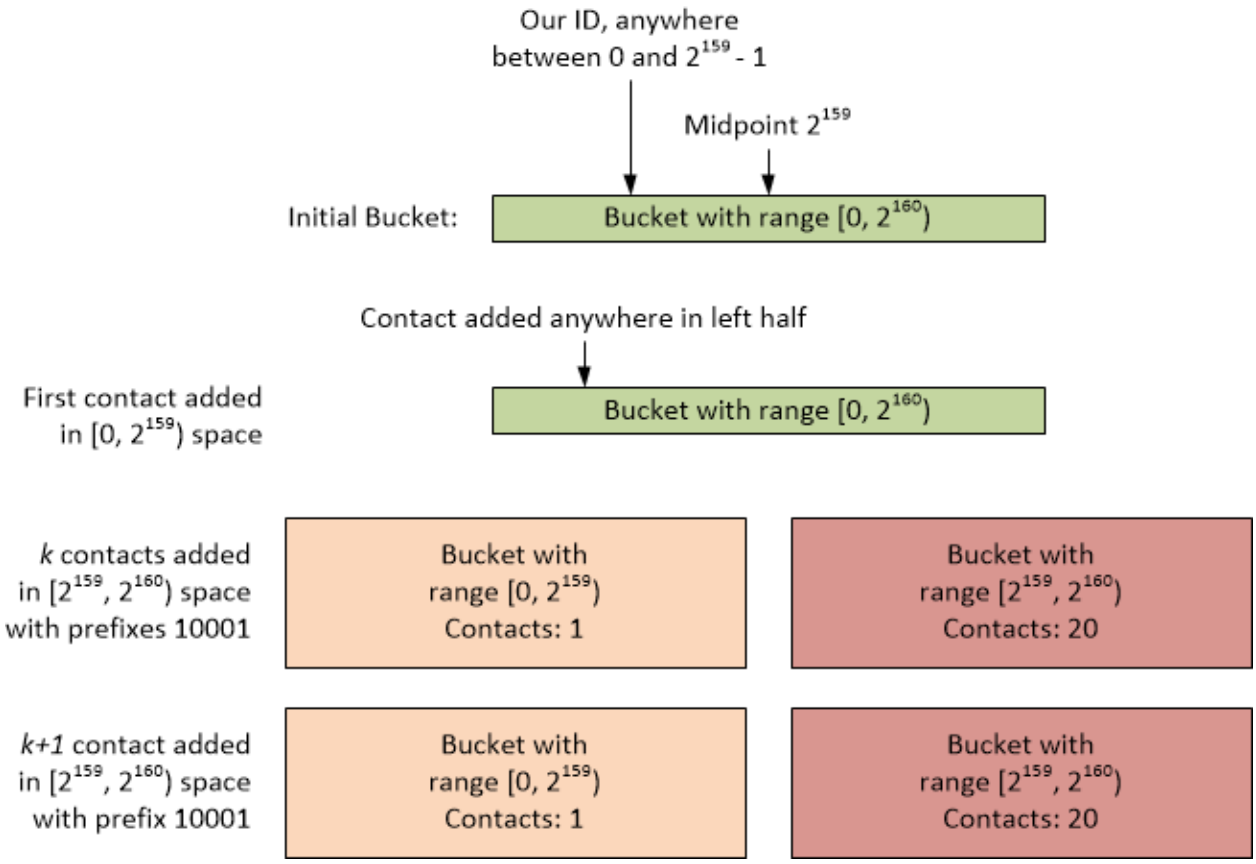

Figure 4: Degrading Adding Contacts

We can verify this with a unit test.

Code Listing 19: Forcing Add Contact Failure

```
[TestMethod]
public void ForceFailedAddTest()
{
  Contact dummyContact = new Contact(new VirtualProtocol(), ID.Zero);
  ((VirtualProtocol)dummyContact.Protocol).Node =
    new Node(dummyContact, new VirtualStorage());


  IBucketList bucketList = SetupSplitFailure();


  Assert.IsTrue(bucketList.Buckets.Count == 2, "Bucket split should have
occurred.");
```

```
  Assert.IsTrue(bucketList.Buckets[0].Contacts.Count == 1,
    "Expected 1 contact in bucket 0.");

  Assert.IsTrue(bucketList.Buckets[1].Contacts.Count == 20,
    "Expected 20 contacts in bucket 1.");


  // This next contact should not split the bucket as depth == 5 and
therefore adding
  // the contact will fail.

  // Any unique ID >= 2^159 will do.

  byte[] id = new byte[20];

  id[19] = 0x80;

  Contact newContact = new Contact(dummyContact.Protocol, new ID(id));

  bucketList.AddContact(newContact);


  Assert.IsTrue(bucketList.Buckets.Count == 2,
    "Bucket split should not have occurred.");

Assert.IsTrue(bucketList.Buckets[0].Contacts.Count == 1,
    "Expected 1 contact in bucket 0.");

Assert.IsTrue(bucketList.Buckets[1].Contacts.Count == 20,
    "Expected 20 contacts in bucket 1.");


  // Verify CanSplit -> Evict did not happen.

  Assert.IsFalse(bucketList.Buckets[1].Contacts.Contains(newContact),
    "Expected new contact NOT to replace an older contact.");

}
```

What we've effectively done is break Kademlia, as the peer will no longer accept half of the possible ID range. As long as the peer ID is outside the range of a bucket whose shared prefix mod **b** is **0**, we can continue this process by adding contacts with a shared prefixes (assume **b==5**) **01xxx**, **001xx**, **0001x**, and **00001**, and again for every multiple of **b** bits. If a peer has a "small" ID, you can easily prevent it from accepting new contacts within half of its bucket ranges.

There are several ways to mitigate this:

- IDs should not be created by the user; they should be assigned by the library. Of course, given the open-source nature of all these implementations, enforcing this is impossible.

- A contact's ID should be unique for its network address—in other words, a malicious peer should not be able to create multiple contacts simply by providing a unique ID in its contact request.
- One might consider increasing b as i in 2^i increases. There might be some justification for this, as the range 2^{159} through $2^{160} - 1$ contains half the possible contacts, one might allow the depth for bucket splitting to be greater than the recommended b = 5.

Implementation

Figure 5 shows the flowchart of what we are initially implementing.

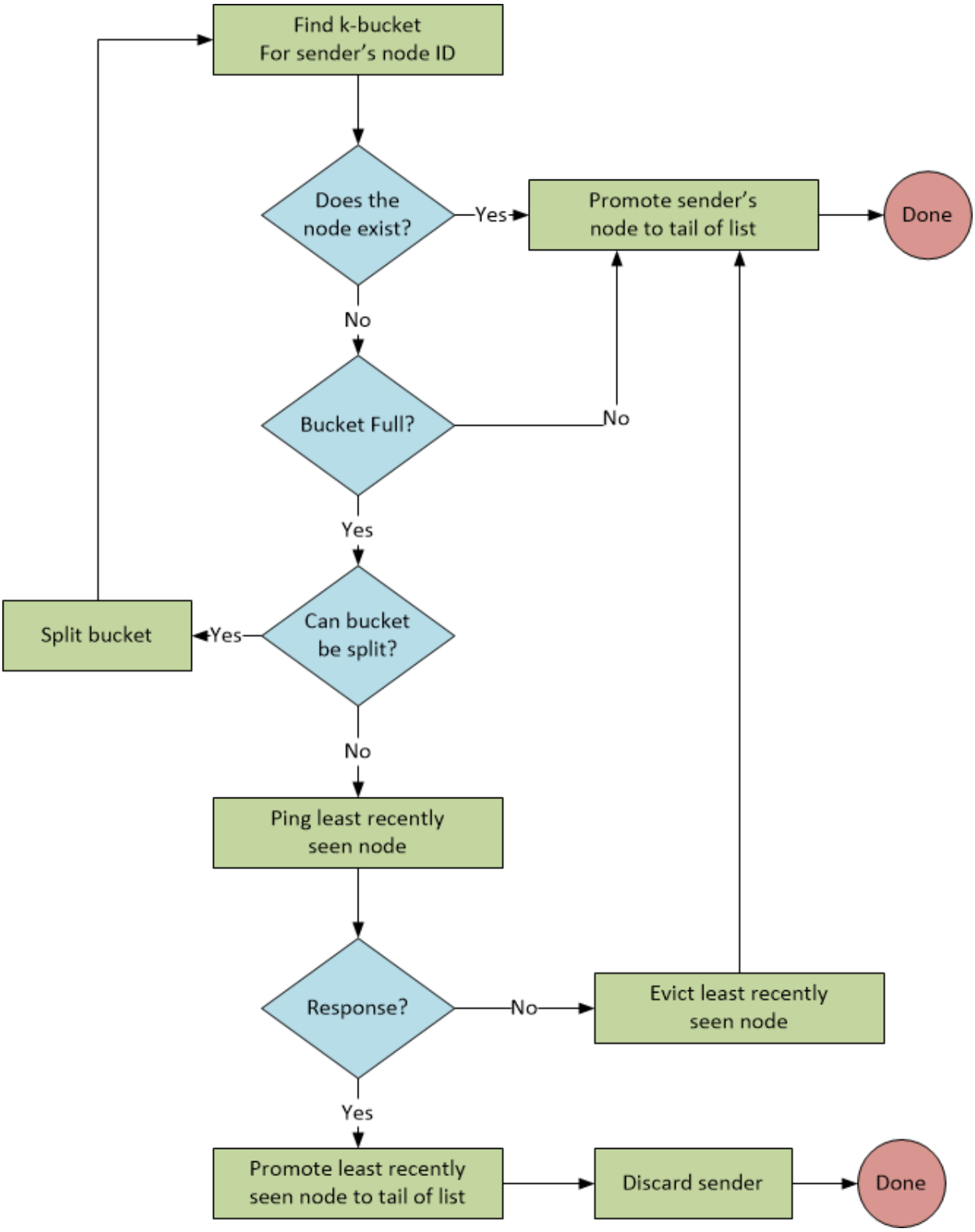

Figure 5: Adding Contacts with Bucket Splitting

As I'm using Brian Muller's implementation as the authority with regards to the spec, we'll go with how he coded the algorithm and will (eventually) incorporate the fallback where we discard nodes in a full k-bucket that don't respond to a ping—but that's later.

The **`BucketList`** class implements the algorithm to add a contact. Note that the **`lock`** statements ensure bucket lists are manipulated synchronously, as the peer server will be receiving commands asynchronously.

Code Listing 20: The AddContact Method Implementation

```
public void AddContact(Contact contact)
{
  Validate.IsFalse<OurNodeCannotBeAContactException>(ourID == contact.ID,
    "Cannot add ourselves as a contact!");
  contact.Touch();            // Update the LastSeen to now.

  lock (this)
  {
    KBucket kbucket = GetKBucket(contact.ID);

    if (kbucket.Contains(contact.ID))
    {
      // Replace the existing contact, updating the network info and
      // LastSeen timestamp.
      kbucket.ReplaceContact(contact);
    }
    else if (kbucket.IsBucketFull)
    {
      if (CanSplit(kbucket))
      {
        // Split the bucket and try again.
         (KBucket k1, KBucket k2) = kbucket.Split();
        int idx = GetKBucketIndex(contact.ID);
        buckets[idx] = k1;
```

```
        buckets.Insert(idx + 1, k2);
        buckets[idx].Touch();
        buckets[idx + 1].Touch();
        AddContact(contact);
      }
      else
      {
        // TODO: Ping the oldest contact to see if it's still
        // around and replace it if not.
      }
    }
    else
    {
      // Bucket isn't full, so just add the contact.
      kbucket.AddContact(contact);
    }
  }
}
```

Later, this will be extended to handle delayed eviction and adding new contacts that can't fit in the bucket to a pending queue.

We have a few helper methods in this class as well.

Code Listing 21: Helper Methods

```
protected virtual bool CanSplit(KBucket kbucket)
{
  lock (this)
  {
    return kbucket.HasInRange(ourID) || ((kbucket.Depth() % Constants.B)
!= 0);
  }
```

```
}

protected int GetKBucketIndex(ID otherID)
{
  lock (this)
  {
    return buckets.FindIndex(b => b.HasInRange(otherID));
  }
}

/// <summary>
/// Returns number of bits that are in common across all contacts.
/// If there are no contacts, or no shared bits, the return is 0.
/// </summary>
public int Depth()
{
  bool[] bits = new bool[0];

  if (contacts.Count > 0)
  {
    // Start with the first contact.
    bits = contacts[0].ID.Bytes.Bits().ToArray();

    contacts.Skip(1).ForEach(c => bits = SharedBits(bits, c.ID));
  }

  return bits.Length;
}

/// <summary>
```

```
/// Returns a new bit array of just the shared bits.
/// </summary>
protected bool[] SharedBits(bool[] bits, ID id)
{
  bool[] idbits = id.Bytes.Bits().ToArray();

  // Useful for viewing the bit arrays.
  //string sbits1 = System.String.Join("", bits.Select(b => b ? "1" :
"0"));
  //string sbits2 = System.String.Join("", idbits.Select(b => b ? "1" :
"0"));

  int q = Constants.ID_LENGTH_BITS - 1;
  int n = bits.Length - 1;
  List<bool> sharedBits = new List<bool>();

  while (n >= 0 && bits[n] == idbits[q])
  {
    sharedBits.Insert(0, (bits[n]));
    --n;
    --q;
  }

  return sharedBits.ToArray();
}
```

The method **`CanSplit`** is **`virtual`**, so you can provide a different implementation.

The majority of the remaining work is done in the **`KBucket`** class.

Code Listing 22: The Split Method

```
public (KBucket, KBucket) Split()
{
  BigInteger midpoint = (Low + High) / 2;
  KBucket k1 = new KBucket(Low, midpoint);
  KBucket k2 = new KBucket(midpoint, High);

  Contacts.ForEach(c =>
  {
    // <, because the High value is exclusive in the HasInRange test.
    KBucket k = c.ID < midpoint ? k1 : k2;
    k.AddContact(c);
  });

  return (k1, k2);
}
```

Recall that the ID is stored as a little-endian value, and the prefix is most significant bits, so we have to work the ID backwards, **`n-1`** to **`0`**. Also note the implementation of the **`Bytes`** property in the ID class:

Code Listing 23: The ID.Bytes Property

```
/// <summary>
/// The array returned is in little-endian order (lsb at index 0)
/// </summary>
public byte[] Bytes
{
  get
  {
    // Zero-pad msb's if ToByteArray length != Constants.LENGTH_BYTES
    byte[] bytes = new byte[Constants.ID_LENGTH_BYTES];
```

```
    byte[] partial =
id.ToByteArray().Take(Constants.ID_LENGTH_BYTES).ToArray();
    // remove msb 0 at index 20.

    partial.CopyTo(bytes, 0);


    return bytes;

  }

}
```

Unit tests

Here are a few basic unit tests. The **`VirtualProtocol`** and **`VirtualStorage`** classes will be discussed later. Also note that the constructors for **`Contact`** don't match the code in the basic framework shown previously, as the unit tests here are reflective of the final implementation of the code base.

Code Listing 24: AddContact Unit Tests

```
[TestMethod]

public void UniqueIDAddTest()

{

Contact dummyContact = new Contact(new VirtualProtocol(), ID.Zero);

  ((VirtualProtocol)dummyContact.Protocol).Node =
      new Node(dummyContact, new VirtualStorage());

  BucketList bucketList = new BucketList(ID.RandomIDInKeySpace,
dummyContact);

  Constants.K.ForEach(() => bucketList.AddContact(
      new Contact(null, ID.RandomIDInKeySpace)));

  Assert.IsTrue(bucketList.Buckets.Count == 1,
      "No split should have taken place.");

  Assert.IsTrue(bucketList.Buckets[0].Contacts.Count == Constants.K,
      "K contacts should have been added.");


}
```

```
[TestMethod]
public void DuplicateIDTest()
{
  Contact dummyContact = new Contact(new VirtualProtocol(), ID.Zero);
  ((VirtualProtocol)dummyContact.Protocol).Node =
      new Node(dummyContact, new VirtualStorage());
  BucketList bucketList = new BucketList(ID.RandomIDInKeySpace,
dummyContact);
  ID id = ID.RandomIDInKeySpace;
  bucketList.AddContact(new Contact(null, id));
  bucketList.AddContact(new Contact(null, id));
  Assert.IsTrue(bucketList.Buckets.Count == 1, "No split should have
taken place.");
  Assert.IsTrue(bucketList.Buckets[0].Contacts.Count == 1,
      "Bucket should have one contact.");
}

[TestMethod]
public void BucketSplitTest()
{
  Contact dummyContact = new Contact(new VirtualProtocol(), ID.Zero);
  ((VirtualProtocol)dummyContact.Protocol).Node =
      new Node(dummyContact, new VirtualStorage());
  BucketList bucketList = new BucketList(ID.RandomIDInKeySpace,
dummyContact);
  Constants.K.ForEach(() => bucketList.AddContact(
      new Contact(null, ID.RandomIDInKeySpace)));
  bucketList.AddContact(new Contact(null, ID.RandomIDInKeySpace));
  Assert.IsTrue(bucketList.Buckets.Count > 1,
      "Bucket should have split into two or more buckets.");
}
```

Distribution tests reveal importance of randomness

What happens instead if we randomize the ID based on a random distribution of *bucket slot* rather than a simple random ID? By this, we mean distributing the IDs evenly across the bucket space, not the ID space. Some helper methods:

Code Listing 25: Randomize Bits Helper Methods

```
public ID RandomizeBeyond(int bit, int minLsb = 0, bool forceBit1 =
false)
{
  byte[] randomized = Bytes;


  ID newid = new ID(randomized);


  // TODO: Optimize
  for (int i = bit + 1; i < Constants.ID_LENGTH_BITS; i++)
  {
    newid.ClearBit(i);
  }


  // TODO: Optimize
  for (int i = minLsb; i < bit; i++)
  {
    if ((rnd.NextDouble() < 0.5) || forceBit1)
    {
      newid.SetBit(i);
    }
  }


  return newid;
}
```

```
/// <summary>
/// Clears the bit n, from the little-endian LSB.
/// </summary>
public ID ClearBit(int n)
{
  byte[] bytes = Bytes;
  bytes[n / 8] &= (byte)((1 << (n % 8)) ^ 0xFF);
  id = new BigInteger(bytes.Append0());

  // for continuations.
  return this;
}

/// <summary>
/// Sets the bit n, from the little-endian LSB.
/// </summary>
public ID SetBit(int n)
{
  byte[] bytes = Bytes;
  bytes[n / 8] |= (byte)(1 << (n % 8));
  id = new BigInteger(bytes.Append0());

  // for continuations.
  return this;
}
```

Also, a random ID generator, as in Code Listing 26.

Code Listing 26: RandomID and RandomIDInKeySpace

```
public static ID RandomIDInKeySpace
{
  get
  {
    byte[] data = new byte[Constants.ID_LENGTH_BYTES];
    ID id = new ID(data);
    // Uniform random bucket index.
    int idx = rnd.Next(Constants.ID_LENGTH_BITS);
    // 0 <= idx <= 159
    // Remaining bits are randomized to get unique ID.
    id.SetBit(idx);
    id = id.RandomizeBeyond(idx);

    return id;
  }
}

/// <summary>
/// Produce a random ID.
/// </summary>
public static ID RandomID
{
  get
  {
    byte[] buffer = new byte[Constants.ID_LENGTH_BYTES];
    rnd.NextBytes(buffer);
```

```
    return new ID(buffer);
  }
}
```

Let's look at what happens when we assign a node ID as one of 2^i where 0 <= i < 160 and add 3,200 integer random contact IDs. Here's the unit test, which outputs the count of contacts added to each node ID in the set of *i*.

Code Listing 27: Distribution with Random IDs

```
[TestMethod]
public void DistributionTestForEachPrefix()
{
  Contact dummyContact = new Contact(new VirtualProtocol(), ID.Zero);
  ((VirtualProtocol)dummyContact.Protocol).Node =
      new Node(dummyContact, new VirtualStorage());
  Random rnd = new Random();
  StringBuilder sb = new StringBuilder();

  160.ForEach((i) =>
  {
    BucketList bucketList =
        new BucketList(new ID(BigInteger.Pow(new BigInteger(2), i)),
dummyContact);

    3200.ForEach(() =>
    {
      Contact contact = new Contact(new VirtualProtocol(), ID.RandomID);
      ((VirtualProtocol)contact.Protocol).Node =
          new Node(contact, new VirtualStorage());
      bucketList.AddContact(contact);
    });
```

```
    int contacts = bucketList.Buckets.Sum(b => b.Contacts.Count);
    sb.Append(i + "," + contacts + CRLF);
  });

  File.WriteAllText("prefixTest.txt", sb.ToString());
}
```

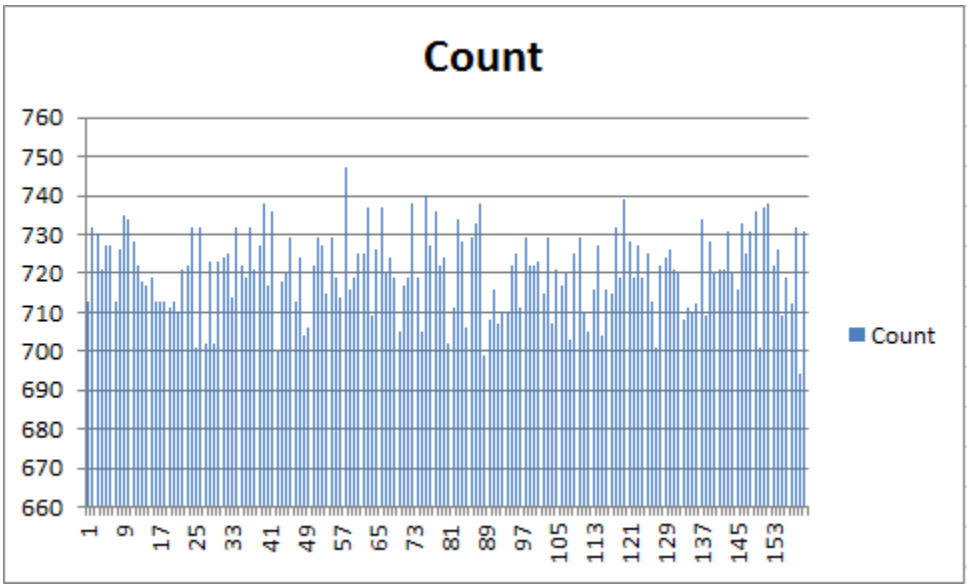

Figure 6: Random ID Distribution

That looks fairly reasonable.

Compare this with the distribution of contact counts when the contact ID is selected from a random prefix with randomized bits after the prefix, as opposed to a random integer ID.

Code Listing 28: Distribution with Random Prefixes

```
[TestMethod]
public void
DistributionTestForEachPrefixWithRandomPrefixDistributedContacts()
{
  Contact dummyContact =
      new Contact(new VirtualProtocol(), ID.Zero);
```

```
  ((VirtualProtocol)dummyContact.Protocol).Node =
      new Node(dummyContact, new VirtualStorage());

  StringBuilder sb = new StringBuilder();


  160.ForEach((i) =>

  {

    BucketList bucketList =
        new BucketList(new ID(BigInteger.Pow(new BigInteger(2), i)),
dummyContact);

    Contact contact = new Contact(new VirtualProtocol(),
ID.RandomIDInKeySpace);

    ((VirtualProtocol)contact.Protocol).Node =
        new Node(contact, new VirtualStorage());

    3200.ForEach(() => bucketList.AddContact(contact));

    int contacts = bucketList.Buckets.Sum(b => b.Contacts.Count);

    sb.Append(i + "," + contacts + CRLF);

  });


  File.WriteAllText("prefixTest.txt", sb.ToString());

}
```

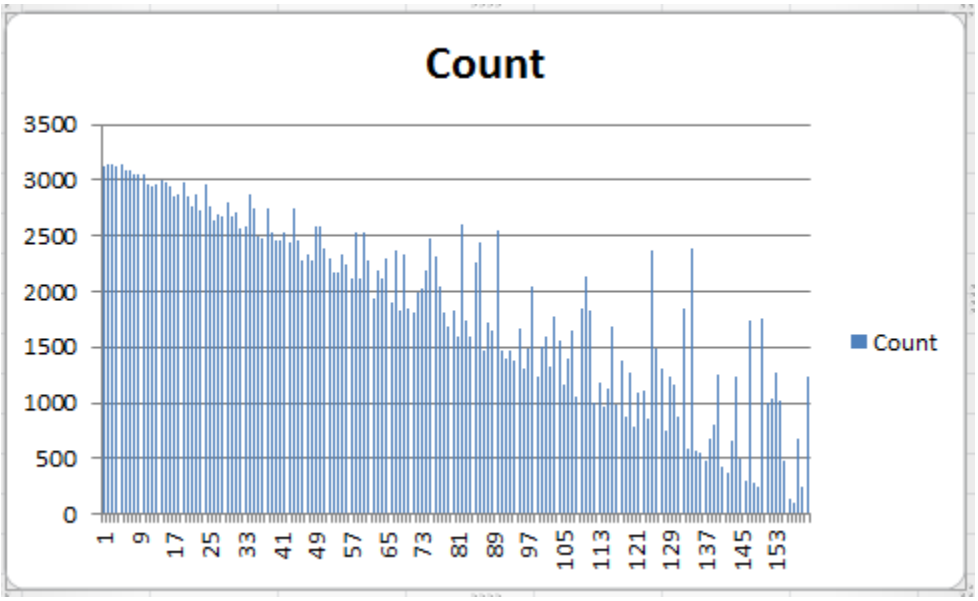

Figure 7: Distributed Prefix Distribution

If there was a question as to whether to choose a node ID based on an even distribution in the prefix space versus simply a random integer ID, I think this clearly demonstrates that a random integer ID is the best choice.

Chapter 5 Node Lookup

From the specification:

"*The most important procedure a Kademlia participant must perform is to locate the k closest nodes to some given node ID. We call this procedure a node lookup. Kademlia employs a recursive algorithm for node lookups. The lookup initiator starts by picking* a *nodes from its closest non-empty k-bucket (or, if that bucket has fewer than a entries, it just takes the a closest nodes it knows of). The initiator then sends parallel, asynchronous FIND_NODE RPCS to the* a *nodes it has chosen,* a *is a system-wide concurrency parameter, such as 3.*"

And:

"*In the recursive step, the initiator resends the FIND_NODE to nodes it has learned about from previous RPCs. (This recursion can begin before all a of the previous RPCs have returned). Of the k nodes the initiator has heard of closest to the target, it picks a that it has not yet queried and resends the FIND_NODE RPC to them. Nodes that fail to respond quickly are removed from consideration until and unless they do respond. If a round of FIND_NODES fails to return a node any closer than the closest already seen, the initiator resends the FIND_NODE to all of the k closest nodes it has not already queried. The lookup terminates when the initiator has queried and gotten responses from the k closest nodes it has seen. When a = 1, the lookup algorithm resembles Chord's in terms of message cost and the latency of detecting failed nodes. However, Kademlia can route for lower latency because it has the flexibility of choosing any one of k nodes to forward a request to.*"

The idea here is to:

- Try really hard to return *k* close nodes.
- Contact other peers in parallel to minimize latency.

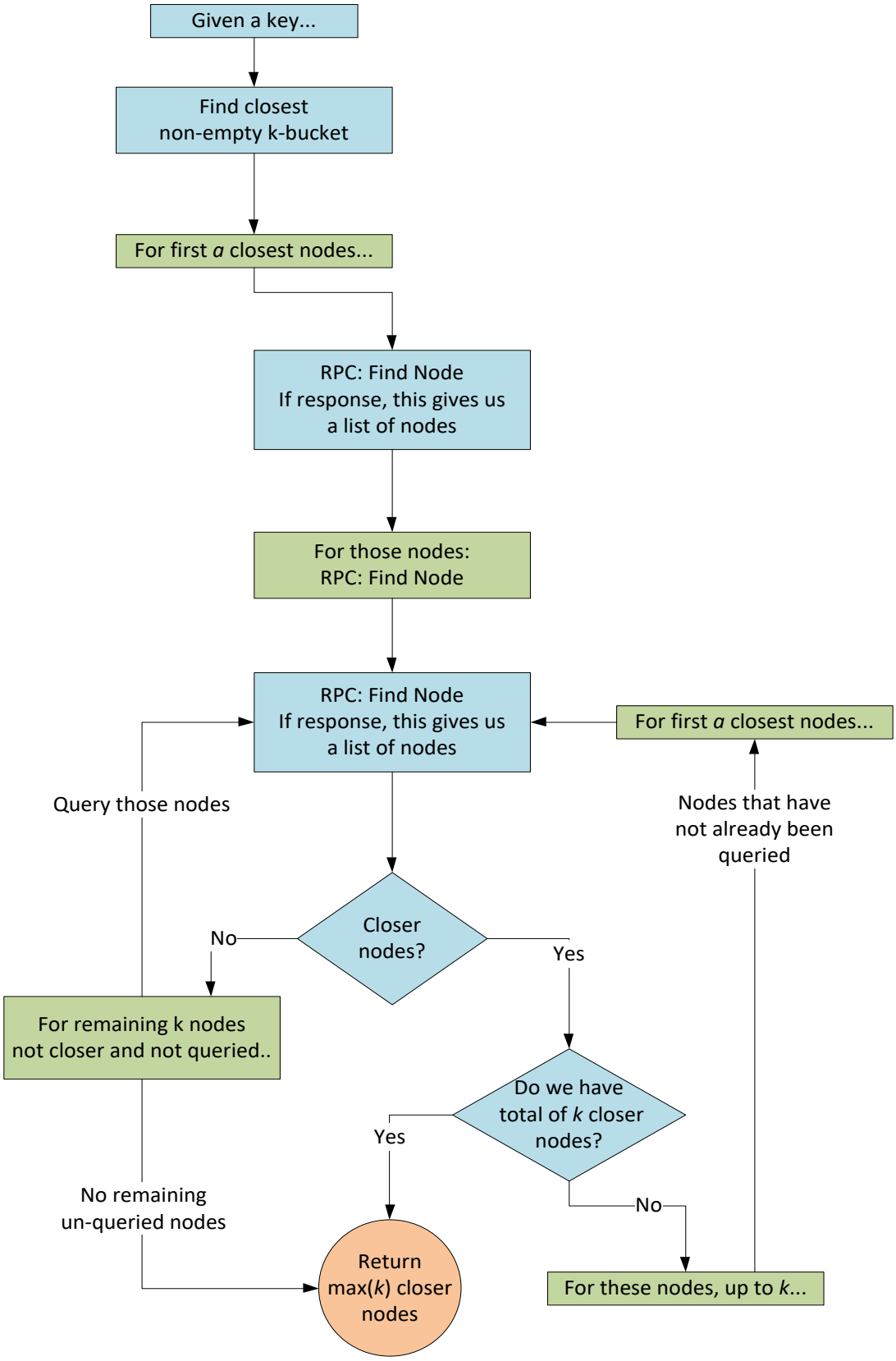

Figure 8: Node Lookup Algorithm

This looks complicated, but the implementation (shown later) is fairly straightforward.

Terminology: The lookup initiator: This is your own peer wanting to make a **`store`** or **`retrieve`** call to other peers. The node lookup is performed before you store or retrieve a value so that your peer has a reasonable.

"*To store a (key,value) pair, a participant locates the k closest nodes to the key and sends them STORE RPCS.*" Here the term "locates" actually means performing the lookup. Contrast with (from the spec): "*To find a (key,value) pair, a node starts by performing a lookup to find the k nodes with IDs closest to the key,*" in which the term "lookup" is specifically used.

We can see this implemented in Brian Muller's Python code.

Code Listing 29: Find Neighbors Algorithm in Python

```
async def get(self, key):
  """
  Get a key if the network has it.
  Returns:
  :class:`None` if not found, the value otherwise.
  """
  dkey = digest(key)
  # if this node has it, return it
  if self.storage.get(dkey) is not None:
    return self.storage.get(dkey)
    node = Node(dkey)
    nearest = self.protocol.router.findNeighbors(node)
    ...
async def set(self, key, value):
  """
  Set the given string key to the given value in the network.
  """
  self.log.debug("setting '%s' = '%s' on network" % (key, value))
  dkey = digest(key)
  return await self.set_digest(dkey, value)
```

```
async def set_digest(self, dkey, value):
  """
  Set the given SHA1 digest key (bytes) to the given value in the
network.
  """
  node = Node(dkey)


  nearest = self.protocol.router.findNeighbors(node)
  ...
```

Here, **`findNeighbors`** is the lookup algorithm described in the specification.

From the spec, what does “nodes that fail to respond quickly” mean? Particularly, the term “quickly”? This is an implementation-specific determination.

What if you, as a peer, don’t have peers in your own k-buckets? That shouldn’t happen (you should at least have the peer you are contacting), but if that peer only has you in its k-buckets, then there’s nothing to return.

From the spec, in “*from its closest non-empty k-bucket,*” what does “closest” mean? I am assuming here that it is the XOR distance metric, but then the question is, what do we use as the “key” for a bucket with a range of contacts? Since this is not defined, the implementation will search all the contacts across all buckets for the initial set of contacts that are closer. Also, the XOR distance computation means that we can’t just ping-pong in an outer search from the bucket containing the range in which the key resides. This better matches the other condition “or, if that bucket has fewer than a entries, it just takes the a closest nodes it knows of,” which implies searching for all **a** closest nodes across all buckets.

Again from the spec: “*The lookup initiator starts by picking a nodes.*” What does “picking” mean? Does this mean additionally sorting the contacts in the “closest bucket” also by closeness? It’s completely undefined.

If you want to try the “closest bucket” version, enable the #define **`TRY_CLOSEST_BUCKET`**, which is implemented like Code Listing 30.

Code Listing 30: Try Closest Bucket

```
KBucket bucket = FindClosestNonEmptyKBucket(key);


// Not in spec -- sort by the closest nodes in the closest bucket.
```

```
List<Contact> nodesToQuery = allNodes.Take(Constants.ALPHA).ToList();
```

Otherwise, the implementation simply gets the closest **a** contacts across all buckets.

Code Listing 31: Closest Contacts Across All Buckets

```
List<Contact> allNodes = node.BucketList.GetCloseContacts(key,
     node.OurContact.ID).Take(Constants.K).ToList();
```

However, this implementation, when testing with virtual nodes (where the system essentially knows every other node) effectively gets the **k** closest contacts because it's searched all the buckets in virtual node space. So, if we want to exercise the algorithm, Code Listing 32 is better.

Code Listing 32: Closest Contacts For Unit Testing

```
List<Contact> allNodes =
    node.BucketList.GetKBucket(key).Contacts.Take(Constants.K).ToList();
```

This actually leads to the next problem. In the initial acquisition of contacts as per the previous code, should contacts (I'm using "contact" and "node" rather interchangeably) that are closer at this point be added to the list of closer contacts? The spec doesn't say not to, but it doesn't explicitly say one should do this. Given that we pick only **a** contacts to start with, we definitely don't have the **k** contacts that the lookup is expected to return, so I'm implementing this as described above—the **a** closest contacts we have are added to the "closer" list, and the **a** farther contacts we have are added to the "farther" list.

Code Listing 33: Adding Closer/Further Contacts to the Probed Contacts List

```
// Also not explicitly in spec:

// Any closer node in the alpha list is immediately added to our closer
contact list,
// and any farther node in the alpha list is immediately added to our
// farther contact list.

closerContacts.AddRange(nodesToQuery.Where(
    n => (n.ID ^ key) < (node.OurContact.ID ^ key)));

fartherContacts.AddRange(nodesToQuery.Where(
    n => (n.ID ^ key) >= (node.OurContact.ID ^ key)));
```

What do we do with the contacts outside of **a**? Given this (from the spec): "*If a round of FIND_NODES fails to return a node any closer than the closest already seen, the initiator resends the FIND_NODE to all of the k closest nodes it has not already queried*," does it apply to the first query of a nodes, or only to the set of nodes returned after the query? I'm going to assume that it applies to the remainder of the a nodes not queried in the first query, which will be a maximum of **k-a** contacts.

The spec says this: "*Most operations are implemented in terms of the above lookup procedure.*" Which operations, and when? We'll have to address this later.

The concept of closeness

From the spec: "*Many of Kademlia's benefits result from its use of a novel XOR metric for distance between points in the key space. XOR is symmetric, allowing Kademlia participants to receive lookup queries from precisely the same distribution of nodes contained in their routing tables.*"

Therefore, the distance between a node and a key is the node ID XORed with the key. Unfortunately, an XOR distance metric is not amenable to a pre-sorted list of IDs. The resulting "distance" computation can be very different for two keys when XOR'd with the contact list. As described on Stack Overflow:[12]

"The thing is that buckets don't have to be full, and if you want to send, let's say, 20 nodes in a response, a single bucket will not suffice. So you must traverse the routing table (either sorted based on your own node ID or by the natural distance) in ascending distance (XOR) order relative to the target key to visit multiple buckets. Because the XOR distance metric folds at each bit-carry (XOR == carry-less addition), it does not map nicely to any routing table layout. In other words, visiting the nearest buckets won't do...I figure that many people simply iterate over the whole routing table, because for regular nodes it will only contain a few dozen buckets at most, and a DHT node does not see much traffic, so it only has to execute this operation a few times per second. If you implement this in a dense, cache-friendly data structure, then the lion's share might actually be the memory traffic and not the CPU instructions doing a few XORs and comparisons.

I.e. a full table scan is just easy to implement."

The author of that post provides an implementation[13] that doesn't require a full table scan, but in my implementation, we'll just do a full scan of the bucket list contacts.

Implementation

Let's start with a baseline implementation that for the moment:

- Doesn't deal with the issue of parallelism.
- Doesn't deal with querying only *a* nodes—we set **a** to the constant k for the moment.
- At the moment, we're also ignoring the fact that this algorithm is the same for node lookup as it is for value lookup.

[12] https://stackoverflow.com/questions/30654398/implementing-find-node-on-torrent-kademlia-routing-table

[13] https://github.com/the8472/mldht/blob/9fb056390b50e9ddf84ed7709283b528a77a0fe5/src/lbms/plugins/mldht/kad/KClosestNodesSearch.java#L104-L170

This simplifies the implementation so that we can provide some unit tests for the basic algorithm, then add the parallelism and *a* concept later, and our unit tests should still pass. Also, the methods here are all marked as **`virtual`** in case you want to override the implementation. First, some helper methods:

Code Listing 34: FindClosestNonEmptyKBucket

```
public virtual KBucket FindClosestNonEmptyKBucket(ID key)
{
  KBucket closest = node.BucketList.Buckets.Where(
      b => b.Contacts.Count > 0).OrderBy(b => b.Key ^
key).FirstOrDefault();

  Validate.IsTrue<NoNonEmptyBucketsException>(closest != null,
      "No non-empty buckets exist.  You must first register a peer and
add
       that peer to your bucketlist.");


  return closest;

}
```

Code Listing 35: GetClosestNodes

```
public List<Contact> GetClosestNodes(ID key, KBucket bucket)
{
  return bucket.Contacts.OrderBy(c => c.ID ^ key).ToList();
}
```

Code Listing 36: RpcFindNodes

```
public
  (List<Contact> contacts, Contact foundBy, string val)
  RpcFindNodes(ID key, Contact contact)
{
  var (newContacts, timeoutError) =
contact.Protocol.FindNode(node.OurContact, key);

  // Null continuation here to support unit tests where a DHT hasn't been
set up.

  dht?.HandleError(timeoutError, contact);
```

```
  return (newContacts, null, null);
}
```

Note that in this code we have a method for handling timeouts and other errors, which we'll describe later.

Code Listing 37: GetCloserNodes

```
public bool GetCloserNodes(

  ID key,

  Contact nodeToQuery,

  Func<
    ID,
    Contact,
    (List<Contact> contacts,
    Contact foundBy,
    string val)> rpcCall,

  List<Contact> closerContacts,

  List<Contact> fartherContacts,

  out string val,

  out Contact foundBy)

{

  // As in, peer's nodes:

  // Exclude ourselves and the peers we're contacting (closerContacts and
  // fartherContacts) to a get unique list of new peers.

  var (contacts, cFoundBy, foundVal) = rpcCall(key, nodeToQuery);

  val = foundVal;

  foundBy = cFoundBy;

  List<Contact> peersNodes = contacts.

    ExceptBy(node.OurContact, c => c.ID).

    ExceptBy(nodeToQuery, c => c.ID).

    Except(closerContacts).

    Except(fartherContacts).ToList();
```

```
  // Null continuation is a special case primarily for unit testing when
we have
  // no nodes in any buckets.
  var nearestNodeDistance = nodeToQuery.ID ^ key;

  lock (locker)
  {
    closerContacts.
      AddRangeDistinctBy(peersNodes.
      Where(p => (p.ID ^ key) < nearestNodeDistance),
      (a, b) => a.ID == b.ID);
  }

  lock (locker)
  {
    fartherContacts.
    AddRangeDistinctBy(peersNodes.
      Where(p => (p.ID ^ key) >= nearestNodeDistance),
      (a, b) => a.ID == b.ID);
  }

  return val != null;
}
```

Note that we always exclude our own node and the nodes we're contacting. Also note the **`lock`** statements to synchronize manipulating the contact list.

Also:

`Func<ID, Contact, (List<Contact> contacts, Contact foundBy, string val)> rpcCall`

This parameter handles calling either **FindNode** or **FindValue**, which we'll discuss later.

This flowchart all comes together in the **Lookup** method in the **Router** class. This is where the complexity of the flowchart is implemented, so there's a lot here. Remember that the **rpcCall** is calling either a **FindNode** or a **FindValue**, so some of the logic here has to figure out to exit the lookup if a value is found. There's extensive use of tuples here as well, which hopefully makes the code clearer! Lastly, this method is actually an abstract method in a base class because later on, we'll see implementing this algorithm as a parallel "find" (something the spec talks about), but for now, Code Listing 38 shows the nonparallel implementation.

Code Listing 38: The Lookup Algorithm

```
public override (bool found, List<Contact> contacts, Contact foundBy,
string val)
  Lookup(

    ID key,

    Func<ID, Contact, (List<Contact> contacts, Contact foundBy, string
val)> rpcCall,

    bool giveMeAll = false)

{

  bool haveWork = true;

  List<Contact> ret = new List<Contact>();

  List<Contact> contactedNodes = new List<Contact>();

  List<Contact> closerContacts = new List<Contact>();

  List<Contact> fartherContacts = new List<Contact>();

  List<Contact> closerUncontactedNodes = new List<Contact>();

  List<Contact> fartherUncontactedNodes = new List<Contact>();


#if DEBUG

  List<Contact> allNodes =
     node.BucketList.GetKBucket(key).Contacts.Take(Constants.K).ToList();

#else

  // This is a bad way to get a list of close contacts with virtual nodes
because
  // we're always going to get the closest nodes right at the get go.
```

```
  List<Contact> allNodes =
    node.BucketList.GetCloseContacts(key,
     node.OurContact.ID).Take(Constants.K).ToList();
#endif

  List<Contact> nodesToQuery = allNodes.Take(Constants.ALPHA).ToList();


  closerContacts.AddRange(nodesToQuery.Where(
    n => (n.ID ^ key) < (node.OurContact.ID ^ key)));
  fartherContacts.AddRange(nodesToQuery.Where(
    n => (n.ID ^ key) >= (node.OurContact.ID ^ key)));


  // The remaining contacts not tested yet can be put here.
  fartherContacts.AddRange(allNodes.Skip(Constants.ALPHA).
    Take(Constants.K - Constants.ALPHA));


  // We're about to contact these nodes.
  contactedNodes.AddRangeDistinctBy(nodesToQuery, (a, b) => a.ID ==
b.ID);


  // Spec: The initiator then sends parallel, asynchronous FIND_NODE RPCS
to the a
  // nodes it has chosen, a is a system-wide concurrency parameter, such
as 3.


  var queryResult =
    Query(key, nodesToQuery, rpcCall, closerContacts, fartherContacts);


  if (queryResult.found)
  {
#if DEBUG       // For unit testing.
    CloserContacts = closerContacts;
```

```
    FartherContacts = fartherContacts;
#endif
  return queryResult;
}


  // Add any new closer contacts to the list we're going to return.
  ret.AddRangeDistinctBy(closerContacts, (a, b) => a.ID == b.ID);


  // Spec: The lookup terminates when the initiator has queried and
received
  // responses from the k closest nodes it has seen.
  while (ret.Count < Constants.K && haveWork)
  {
    closerUncontactedNodes =
closerContacts.Except(contactedNodes).ToList();
    fartherUncontactedNodes =
fartherContacts.Except(contactedNodes).ToList();
    bool haveCloser = closerUncontactedNodes.Count > 0;
    bool haveFarther = fartherUncontactedNodes.Count > 0;


    haveWork = haveCloser || haveFarther;


    // Spec:  Of the k nodes the initiator has heard of closest to the
target...
    if (haveCloser)
    {
      // Spec: ...it picks a that it has not yet queried and resends the
      // FIND_NODE RPC to them.
      var newNodesToQuery =
closerUncontactedNodes.Take(Constants.ALPHA).ToList();
```

```
      // We're about to contact these nodes.
      contactedNodes.AddRangeDistinctBy(newNodesToQuery, (a, b) => a.ID
== b.ID);

      queryResult =
        Query(key, newNodesToQuery, rpcCall, closerContacts,
fartherContacts);

      if (queryResult.found)
      {
#if DEBUG       // For unit testing.
        CloserContacts = closerContacts;
        FartherContacts = fartherContacts;
#endif
        return queryResult;
      }
    }
    else if (haveFarther)
    {
      var newNodesToQuery =
fartherUncontactedNodes.Take(Constants.ALPHA).ToList();

      // We're about to contact these nodes.
      contactedNodes.AddRangeDistinctBy(fartherUncontactedNodes,
        (a, b) => a.ID == b.ID);

      queryResult =
        Query(key, newNodesToQuery, rpcCall, closerContacts,
fartherContacts);

      if (queryResult.found)
```

```
      {
#if DEBUG       // For unit testing.
        CloserContacts = closerContacts;
        FartherContacts = fartherContacts;
#endif
        return queryResult;
      }
    }
  }

#if DEBUG       // For unit testing.
  CloserContacts = closerContacts;
  FartherContacts = fartherContacts;
#endif

  // Spec (sort of): Return max(k) closer nodes, sorted by distance.
  // For unit testing, giveMeAll can be true so that we can match against
our
  // alternate way of getting closer contacts.
  return (
    false,
    (giveMeAll ? ret : ret.Take(Constants.K).OrderBy(c => c.ID ^
key).ToList()),
    null,
    null);
}
```

This algorithm is iteration, not recursion, as the specification states. Every implementation I've seen uses iteration.

Here's an extension method that is used.

Code Listing 39: AddRangeDistinctBy

```
public static void AddRangeDistinctBy<T>(
  this List<T> target,
  IEnumerable<T> src, Func<T, T, bool> equalityComparer)
{
  src.ForEach(item =>
  {
    // no items in the list must match the item.
    if (target.None(q => equalityComparer(q, item)))
    {
      target.Add(item);
    }
  });
}
```

Unit tests

Before getting into unit tests, we need to be able to create virtual nodes, which means implementing a minimal **`VirtualProtocol`**:

Code Listing 40: Virtual Node

```
public class VirtualProtocol : IProtocol
{
  public Node Node { get; set; }

  /// <summary>
  /// For unit testing with deferred node setup.
  /// </summary>
  public VirtualProtocol(bool responds = true)
  {
    Responds = responds;
  }
```

```
  /// <summary>
  /// Register the in-memory node with our virtual protocol.
  /// </summary>
  public VirtualProtocol(Node node, bool responds = true)
  {
    Node = node;
    Responds = responds;
  }

  public RpcError Ping(Contact sender)
  {
    return new RpcError() { TimeoutError = !Responds };
  }

  /// <summary>
  /// Get the list of contacts for this node closest to the key.
  /// </summary>
  public (List<Contact> contacts, RpcError error) FindNode(Contact
sender, ID key)
  {
    return (Node.FindNode(sender, key).contacts, NoError());
  }

  /// <summary>
  /// Returns either contacts or null if the value is found.
  /// </summary>
  public (List<Contact> contacts, string val, RpcError error)
    FindValue(Contact sender, ID key)
```

```
  {
    var (contacts, val) = Node.FindValue(sender, key);

    return (contacts, val, NoError());
  }

  /// <summary>
  /// Stores the key-value on the remote peer.
  /// </summary>
  public RpcError Store(
    Contact sender,
    ID key,
    string val,
    bool isCached = false,
    int expTimeSec = 0)
  {
    Node.Store(sender, key, val, isCached, expTimeSec);

    return NoError();
  }

  protected RpcError NoError()
  {
    return new RpcError();
  }
}
```

Notice that for unit testing, we have the option to simulate a nonresponding node.

We also have to implement **`FindNode`** in the **`Node`** class (the other methods we'll implement later).

Code Listing 41: FindNode

```
public (List<Contact> contacts, string val) FindNode(Contact sender, ID
key)

{

  Validate.IsFalse<SendingQueryToSelfException>(sender.ID ==
ourContact.ID,
      "Sender should not be ourself!");

  SendKeyValuesIfNewContact(sender);

  bucketList.AddContact(sender);


  // Exclude sender.

  var contacts = bucketList.GetCloseContacts(key, sender.ID);


  return (contacts, null);

}
```

Note that this call attempts to add the sender as a contact, which either adds (maybe) the contact to the recipient's bucket list or updates the contact information. If the contact is new, any key-values that are "closer" to the new peer are sent. This is an important aspect of the Kademlia algorithm: peers that are "closer" to the stored key-values get those key-values so that the efficiency of lookups to find a key-value is improved.

We also need an algorithm for finding close contacts across the recipient's bucket range.

Code Listing 42: GetCloseContacts

```
/// <summary>

/// Brute force distance lookup of all known contacts, sorted by
distance, then we
/// take at most k (20) of the closest.

/// </summary>

/// <param name="toFind">The ID for which we want to find close
contacts.</param>

/// <param name="exclude">The ID to exclude (the requestor's ID)</param>

public List<Contact> GetCloseContacts(ID key, ID exclude)
```

```
{
  lock (this)
  {
    var contacts = buckets.
    SelectMany(b => b.Contacts).
      Where(c => c.ID != exclude).
      Select(c => new { contact = c, distance = c.ID ^ key }).
      OrderBy(d => d.distance).
      Take(Constants.K);

    return contacts.Select(c => c.contact).ToList();
  }
}
```

This tests the sorting and the maximum number of contacts returned limit. Note that it's probably not the most ideal situation that I'm using all random IDs; however, to ensure consistent unit testing, we seed the random number generator with the same value in **DEBUG** mode.

Code Listing 43: Debug Mode Seeding of Random Values

```
#if DEBUG
  public static Random rnd = new Random(1);
#else
  private static Random rnd = new Random();
#endif
```

GetCloseContacts unit test

Finally, we can implement the "get close contacts" unit test.

Code Listing 44: GetCloseContactsOrderedTest()

```
[TestMethod]
```

```
public void GetCloseContactsOrderedTest()
{
  Contact sender = new Contact(null, ID.RandomID);
  Node node = new Node(new Contact(null, ID.RandomID), new
VirtualStorage());
  List<Contact> contacts = new List<Contact>();
  // Force multiple buckets.
  100.ForEach(() => contacts.Add(new Contact(null, ID.RandomID)));
  contacts.ForEach(c => node.BucketList.AddContact(c));
  ID key = ID.RandomID;            // Pick an ID
  List<Contact> closest = node.FindNode(sender, key).contacts;

  Assert.IsTrue(closest.Count == Constants.K, "Expected K contacts to be
returned.");

  // The contacts should be in ascending order with respect to the key.
  var distances = closest.Select(c => c.ID ^ key).ToList();
  var distance = distances[0];

  // Verify distances are in ascending order:
  distances.Skip(1).ForEach(d =>
  {
    Assert.IsTrue(distance < d, "Expected contacts ordered by
distance.");
    distance = d;
  });

  // Verify the contacts with the smallest distances were returned from
all
  // possible distances.
```

```
  var lastDistance = distances[distances.Count - 1];

  var others = node.BucketList.Buckets.SelectMany(
    b => b.Contacts.Except(closest)).Where(c => (c.ID ^ key) <
lastDistance);

  Assert.IsTrue(others.Count() == 0,
     "Expected no other contacts with a smaller distance than the
greatest
      distance to exist.");

}
```

NoNodesToQuery unit test

This test simply verifies that we get no new nodes to query given that all the nodes we're contacting are already being contacted.

Code Listing 45: NoNodes ToQuery unit test

```
/// <summary>

/// Given that all the nodes we're contacting are nodes *being*
contacted,

/// the result should be no new nodes to contact.

/// </summary>

[TestMethod]

public void NoNodesToQueryTest()

{

  // Setup

  router = new Router(new Node(new Contact(null, ID.Mid), new
VirtualStorage()));


  nodes = new List<Node>();

  20.ForEach((n) => nodes.Add(new Node(new Contact(null,
    new ID(BigInteger.Pow(new BigInteger(2), n))), new
VirtualStorage())));
```

```
  // Fixup protocols:
  nodes.ForEach(n => n.OurContact.Protocol = new VirtualProtocol(n));


  // Our contacts:
  nodes.ForEach(n => router.Node.BucketList.AddContact(n.OurContact));


  // Each peer needs to know about the other peers except of course
itself.
  nodes.ForEach(n => nodes.Where(nOther => nOther != n).
    ForEach(nOther => n.BucketList.AddContact(nOther.OurContact)));
  // Select the key such that n ^ 0 == n
  // This ensures that the distance metric uses only the node ID, which
makes for
  // an integer difference for distance, not an XOR distance.
  key = ID.Zero;
  // all contacts are in one bucket.
  contactsToQuery = router.Node.BucketList.Buckets[0].Contacts;
  closerContacts = new List<Contact>();
  fartherContacts = new List<Contact>();


  contactsToQuery.ForEach(c =>
  {
    router.GetCloserNodes(key, c, router.RpcFindNodes,
      closerContacts,
      fartherContacts,
      out var _, out var _);
  });


  Assert.IsTrue(closerContacts.ExceptBy(contactsToQuery, c=>c.ID).Count()
== 0,
      "No new nodes expected.");
```

```
  Assert.IsTrue(fartherContacts.ExceptBy(contactsToQuery,
c=>c.ID).Count() == 0,
      "No new nodes expected.");

}
```

Lookup unit test

Ironically, there really is no alternative way of getting these nodes, particularly with regards to the **`while`** loop in the **`Lookup`** method. So without beating my head over the issue, I implemented this test in Code Listing 46.

Code Listing 46: Lookup Unit Test

```
public void LookupTest()

{

  // Seed with different random values

  100.ForEach(seed =>

  {

    ID.rnd = new Random(seed);

    Setup();


    List<Contact> closeContacts =
      router.Lookup(key, router.RpcFindNodes, true).contacts;

    List<Contact> contactedNodes = new List<Contact>(closeContacts);


    // Is the above call returning the correct number of close contacts?

    // The unit test for this is sort of lame.  We should get at least
    // as many contacts as when calling GetCloserNodes.


    GetAltCloseAndFar(
      contactsToQuery,
      closerContactsAltComputation,
      fartherContactsAltComputation);
```

```
    Assert.IsTrue(closeContacts.Count >=
closerContactsAltComputation.Count,
       "Expected at least as many contacts.");


    // Technically, we can't even test whether the contacts returned
    // in GetCloserNodes exists in router.Lookup because it may have
found nodes
    // even closer, and it only returns K nodes!

    // We can overcome this by eliminating the Take in the return of
router.Lookup().


    closerContactsAltComputation.ForEach(c =>
      Assert.IsTrue(closeContacts.Contains(c)));

  });

}
```

The setup for this unit test is complex.

Code Listing 47: Lookup Setup

```
protected void Setup()

{

  // Setup

  router = new Router(
    new Node(new Contact(null, ID.RandomID), new VirtualStorage()));


  nodes = new List<Node>();

  100.ForEach(() =>

  {

    Contact contact = new Contact(new VirtualProtocol(), ID.RandomID);

    Node node = new Node(contact, new VirtualStorage());

    ((VirtualProtocol)contact.Protocol).Node = node;

    nodes.Add(node);
```

```
  });


  // Fixup protocols:

  nodes.ForEach(n => n.OurContact.Protocol = new VirtualProtocol(n));


  // Our contacts:

  nodes.ForEach(n => router.Node.BucketList.AddContact(n.OurContact));


  // Each peer needs to know about the other peers except of course itself.

  nodes.ForEach(n => nodes.Where(nOther => nOther != n).
    ForEach(nOther =>n.BucketList.AddContact(nOther.OurContact)));

  // Pick a random bucket, or bucket where the key is in range, otherwise
  // we're defeating the purpose of the algorithm.

  key = ID.RandomID;            // Pick an ID

  // DO NOT DO THIS:

  // List<Contact> nodesToQuery = router.Node.BucketList.GetCloseContacts(
  // key, router.Node.OurContact.ID).Take(Constants.ALPHA).ToList();


  contactsToQuery =

router.Node.BucketList.GetKBucket(key).Contacts.Take(Constants.ALPHA).ToLis
t();

  // or:

  // contactsToQuery =
  //
router.FindClosestNonEmptyKBucket(key).Contacts.Take(Constants.ALPHA).ToLis
t();


  closerContacts = new List<Contact>();

  fartherContacts = new List<Contact>();


  closerContactsAltComputation = new List<Contact>();
```

```
  fartherContactsAltComputation = new List<Contact>();

  theNearestContactedNode = contactsToQuery.OrderBy(n => n.ID ^
key).First();

  distance = theNearestContactedNode.ID.Value ^ key.Value;

}
```

Furthermore, we use a different implementation for acquiring closer and farther contacts, so that we can verify the algorithm under test. Code Listing 48 shows the alternate implementation.

Code Listing 48: Alternate Implementation for Getting Closer and Farther Nodes

```
protected void GetAltCloseAndFar(
  List<Contact> contactsToQuery,
  List<Contact> closer, List<Contact> farther)

{

  // For each node (ALPHA == K for testing) in our bucket (nodesToQuery)
we're
  // going to get k nodes closest to the key:

  foreach (Contact contact in contactsToQuery)

  {

    // Find the node we're contacting:

    Node contactNode = nodes.Single(n => n.OurContact == contact);


    // Close contacts except ourself and the nodes we're contacting.

    // Note that of all the contacts in the bucket list, many of the k
returned

    // by the GetCloseContacts call are contacts we're querying, so they
are
    // being excluded.

    var closeContactsOfContactedNode =

      contactNode.

      BucketList.

      GetCloseContacts(key, router.Node.OurContact.ID).

      ExceptBy(contactsToQuery, c => c.ID.Value);
```

```
    foreach (Contact closeContactOfContactedNode in
closeContactsOfContactedNode)
    {
      // Which of these contacts are closer?
      if ((closeContactOfContactedNode.ID ^ key) < distance)
      {
        closer.AddDistinctBy(closeContactOfContactedNode, c =>
c.ID.Value);
      }

      // Which of these contacts are farther?
      if ((closeContactOfContactedNode.ID ^ key) >= distance)
      {
        farther.AddDistinctBy(closeContactOfContactedNode, c =>
c.ID.Value);
      }
    }
  }
}
```

SimpleCloserContacts test

This test, and the next one, exercise the part of the **Lookup** algorithm before the while loop. Note how the bucket and IDs are set up.

Code Listing 49: SimpleAllCloserContactsTest

```
[TestMethod]
public void SimpleAllCloserContactsTest()
{
  // Setup
```

```
  // By selecting our node ID to zero, we ensure that all distances of
  // other nodes are > the distance to our node.

  router = new Router(new Node(new Contact(null, ID.Max), new
VirtualStorage()));


  nodes = new List<Node>();

  Constants.K.ForEach((n) => nodes.Add(new Node(new Contact(null,
    new ID(BigInteger.Pow(new BigInteger(2), n))), new
VirtualStorage())));


  // Fixup protocols:

  nodes.ForEach(n => n.OurContact.Protocol = new VirtualProtocol(n));


  // Our contacts:

  nodes.ForEach(n => router.Node.BucketList.AddContact(n.OurContact));


  // Each peer needs to know about the other peers except of course
itself.

  nodes.ForEach(n => nodes.Where(nOther => nOther != n).
    ForEach(nOther => n.BucketList.AddContact(nOther.OurContact)));


  // Select the key such that n ^ 0 == n

  // This ensures that the distance metric uses only the node ID, which
makes
  // for an integer difference for distance, not an XOR distance.

  key = ID.Zero;

  // all contacts are in one bucket.

  contactsToQuery = router.Node.BucketList.Buckets[0].Contacts;


  var contacts = router.Lookup(key, router.RpcFindNodes, true).contacts;
```

```
  Assert.IsTrue(contacts.Count == Constants.K, "Expected k closer
contacts.");

  Assert.IsTrue(router.CloserContacts.Count == Constants.K,
    "All contacts should be closer.");

  Assert.IsTrue(router.FartherContacts.Count == 0,
    "Expected no farther contacts.");

}
```

SimpleFartherContacts test

Again, note how the bucket and IDs are set up in Code Listing 50.

Code Listing 50: SimpleAllFartherContactsTest

```
/// <summary>

/// Creates a single bucket with node IDs 2^i for i in [0, K) and

/// 1. use a key with ID.Value == 0 to that distance computation is an
/// integer difference

/// 2. use an ID.Value == 0 for our node ID so all other nodes are
farther.

/// </summary>

[TestMethod]

public void SimpleAllFartherContactsTest()

{

  // Setup

  // By selecting our node ID to zero, we ensure that all distances of
other
  // nodes are > the distance to our node.

  router = new Router(new Node(new Contact(null, ID.Zero), new
VirtualStorage()));


  nodes = new List<Node>();
```

```
  Constants.K.ForEach((n) => nodes.Add(new Node(new Contact(null,
    new ID(BigInteger.Pow(new BigInteger(2), n))), new
VirtualStorage())));


  // Fixup protocols:

  nodes.ForEach(n => n.OurContact.Protocol = new VirtualProtocol(n));


  // Our contacts:

  nodes.ForEach(n => router.Node.BucketList.AddContact(n.OurContact));


  // Each peer needs to know about the other peers except of course
itself.

  nodes.ForEach(n => nodes.Where(nOther => nOther != n).
    ForEach(nOther => n.BucketList.AddContact(nOther.OurContact)));


  // Select the key such that n ^ 0 == n

  // This ensures that the distance metric uses only the node ID, which
makes
  // for an integer difference for distance, not an XOR distance.

  key = ID.Zero;


  var contacts = router.Lookup(key, router.RpcFindNodes, true).contacts;


  Assert.IsTrue(contacts.Count == 0, "Expected no closer contacts.");

  Assert.IsTrue(router.CloserContacts.Count == 0,
    "Did not expected closer contacts.");

  Assert.IsTrue(router.FartherContacts.Count == Constants.K,
    "All contacts should be farther.");

}
```

Chapter 6 Value Lookup

From the spec: "*FIND_VALUE behaves like FIND_NODE - returning (IP address, UDP port, Node ID) triples - with one exception. If the RPC recipient has received a STORE RPC for the key, it just returns the stored value.*"

That seems clear enough, but we must consider this part of the spec as well:

"*To find a (key,value) pair, a node starts by performing a lookup to find the k nodes with IDs closest to the key. However, value lookups use FIND_VALUE rather than FIND_NODE RPCS. Moreover, the procedure halts immediately when any node returns the value. For caching purposes, once a lookup succeeds, the requesting node stores the (key,value) pair at the closest node it observed to the key that did not return the value.*"

However, the spec says this: "*Most operations are implemented in terms of the above lookup procedure.*" When we're performing a lookup, the initiator will be using the lookup call for both finding nodes and finding values. If it's finding values, it needs to stop if/when the value is found.

This statement from the spec can be ambiguous: "*For caching purposes, once a lookup succeeds, the requesting node stores the (key,value) pair at the closest node it observed to the key that did not return the value.*" What does "requesting node" mean? Is it the node performing the lookup, or the node that made the **`GetValue`** request? It would seem to be the former, because "*it observed to the key that did not return the value*" would otherwise not make any sense.

Going back to the code:

```
Func<ID, Contact, (List<Contact> contacts, Contact foundBy, string val)>
rpcCall
```

We can see now the reason for passing in the RPC, as we want to use the exact same lookup algorithm for **`FindNode`** as we do for **`FindValue`**.

Discussing value lookup tests doesn't make sense outside of the context of the **`Dht`** wrapper, so the unit tests for the value lookup will be done in the **`Dht`** testing.

Implementation

First, we need to implement a simple virtual (in memory) storage mechanism. We don't show the implementation of the full interface, as that isn't required yet.

Code Listing 51: Basic Virtual Storage

```
public class VirtualStorage : IStorage
```

```
{
  protected ConcurrentDictionary<BigInteger, StoreValue> store;

  public VirtualStorage()
  {
    store = new ConcurrentDictionary<BigInteger, StoreValue>();
  }

  public bool Contains(ID key)
  {
    return store.ContainsKey(key.Value);
  }

  public string Get(ID key)
  {
    return store[key.Value].Value;
  }

  public string Get(BigInteger key)
  {
    return store[key].Value;
  }
}
```

We can also now implement **`Store`** and **`FindValue`** in the **`Node`** class.

Code Listing 52: Node Store

```
public void Store(
  Contact sender,
  ID key,
  string val,
```

```
  bool isCached = false,
  int expirationTimeSec = 0)
{
  Validate.IsFalse<SendingQueryToSelfException>(sender.ID ==
ourContact.ID,
    "Sender should not be ourself!");
  bucketList.AddContact(sender);

  if (isCached)
  {
    cacheStorage.Set(key, val, expirationTimeSec);
  }
  else
  {
    SendKeyValuesIfNewContact(sender);
    storage.Set(key, val, Constants.EXPIRATION_TIME_SECONDS);
  }
}
```

Code Listing 53: Node FindValue

```
public (List<Contact> contacts, string val) FindValue(Contact sender, ID
key)
{
  Validate.IsFalse<SendingQueryToSelfException>(sender.ID ==
ourContact.ID,
    "Sender should not be ourself!");
  SendKeyValuesIfNewContact(sender);
  bucketList.AddContact(sender);

  if (storage.Contains(key))
  {
```

```
    return (null, storage.Get(key));
  }
  else if (CacheStorage.Contains(key))
  {
    return (null, CacheStorage.Get(key));
  }
  else
  {
    // Exclude sender.
    return (bucketList.GetCloseContacts(key, sender.ID), null);
  }
}
```

Chapter 7 The DHT Class

We use a wrapper **Dht** class, which will become the main entry point for our peer, for interacting with other peers. The purposes of this class are:

- When storing a value, use the **lookup** algorithm to find other closer peers to propagate the key-value.
- When looking up a value, if our peer doesn't have the value, we again use the **lookup** algorithm to find other closer nodes that might have the value.
- Later, we'll add a bootstrapping method that registers our peer with another peer and initializes our bucket list with that peer's closest contacts.

Implementation

Code Listing 54: The Dht Class

```
public class Dht
{
  public BaseRouter Router { get { return router; } }
  public IProtocol Protocol { get { return protocol; } }
  public IStorage OriginatorStorage { get { return originatorStorage; } }

  public Contact Contact { get { return ourContact; } }


  protected BaseRouter router;

  protected IStorage originatorStorage;

  protected IProtocol protocol;

  protected Node node;

  public Dht(
    ID id,
    IProtocol protocol,
    Func<IStorage> storageFactory,
    BaseRouter router)

    {

      originatorStorage = storageFactory();

      FinishInitialization(id, protocol, router);

    }
```

```
  protected void FinishInitialization(ID id, IProtocol protocol,
BaseRouter router)
  {
    ourId = id;
    ourContact = new Contact(protocol, id);
    node = new Node(ourContact);
    node.Dht = this;
    node.BucketList.Dht = this;
    this.protocol = protocol;
    this.router = router;
    this.router.Node = node;
    this.router.Dht = this;
  }
  public void Store(ID key, string val)
  {
    TouchBucketWithKey(key);

    // We're storing to k closer contacts.
    originatorStorage.Set(key, val);
    StoreOnCloserContacts(key, val);
  }

  public (bool found, List<Contact> contacts, string val) FindValue(ID
key)
  {
    TouchBucketWithKey(key);

    string ourVal;
    List<Contact> contactsQueried = new List<Contact>();
```

```
    (bool found, List<Contact> contacts, string val) ret = (false, null,
null);


    if (originatorStorage.TryGetValue(key, out ourVal))
    {
      // Sort of odd that we are using the key-value store to find
      // something the key-value that we originate.
      ret = (true, null, ourVal);
    }
    // else this is where we will deal with republish and cache storage
later
    else
    {
      var lookup = router.Lookup(key, router.RpcFindValue);


      if (lookup.found)
      {
        ret = (true, null, lookup.val);
        // Find the first close contact (other than the one the value
        // was found by) in which to *cache* the key-value.
        var storeTo = lookup.contacts.Where(c => c != lookup.foundBy).
          OrderBy(c => c.ID ^ key).FirstOrDefault();


        if (storeTo != null)
        {
          int separatingNodes = GetSeparatingNodesCount(ourContact,
storeTo);
          RpcError error = storeTo.Protocol.Store(
            node.OurContact,
            key,
            lookup.val,
            true,
            expTimeSec);
```

```
          HandleError(error, storeTo);
        }
      }
    }

  return ret;
  }
}
```

What exactly should the sender do when a value is not found? The **Dht** returns the nearest nodes, but given that the **lookup** failed to find the value, we know these nodes also do not have the value. As far as I've been able to determine, neither the spec nor a search of the web indicates what to do.

Unit tests

LocalStoreFoundValue

To get started, let's just make sure we can set and get values in our local store with an empty bucket list.

Code Listing 55: LocalStoreFindValueTest

```
[TestMethod]
public void LocalStoreFoundValueTest()
{
  VirtualProtocol vp = new VirtualProtocol();
  Dht dht = new Dht(ID.RandomID, vp, () => new VirtualStorage(), new
Router());
  vp.Node = dht.Router.Node;
  ID key = ID.RandomID;
  string val = "Test";
  dht.Store(key, val);
```

```
  string retval = dht.FindValue(key).val;
  Assert.IsTrue(retval == val, "Expected to get back what we stored");
}
```

ValueStoredInCloserNode

This test creates a single contact and stores the value in that contact. We set up the IDs so that the contact's **`ID`** is less (XOR metric) than our peer's **`ID`**, and we use a key of **`ID.Zero`** to prevent further complexities when computing the distance. Most of the code here is to set up the conditions to make this test!

Code Listing 56: ValueStoredInCloserNodeTest

```
[TestMethod]
public void ValueStoredInCloserNodeTest()
{
  VirtualProtocol vp1 = new VirtualProtocol();
  VirtualProtocol vp2 = new VirtualProtocol();
  VirtualStorage store1 = new VirtualStorage();
  VirtualStorage store2 = new VirtualStorage();

  // Ensures that all nodes are closer, because ID.Max ^ n < ID.Max when
n > 0.
  Dht dht = new Dht(ID.Max, vp1, new Router(), store1, store1, new
VirtualStorage());
  vp1.Node = dht.Router.Node;

  ID contactID = ID.Mid;      // a closer contact.
  Contact otherContact = new Contact(vp2, contactID);
  Node otherNode = new Node(otherContact, store2);
  vp2.Node = otherNode;

  // Add this other contact to our peer list.
```

```
  dht.Router.Node.BucketList.AddContact(otherContact);


  // We want an integer distance, not an XOR distance.
  ID key = ID.Zero;


  // Set the value in the other node, to be discovered by the lookup
process.
  string val = "Test";
  otherNode.SimpleStore(key, val);


  Assert.IsFalse(store1.Contains(key),
    "Expected our peer to NOT have cached the key-value.");
  Assert.IsTrue(store2.Contains(key),
    "Expected other node to HAVE cached the key-value.");


  // Try and find the value, given our Dht knows about the other contact.
  string retval = dht.FindValue(key).val;


  Assert.IsTrue(retval == val, "Expected to get back what we stored");
}
```

The method **SimpleStore** simply stores the value in the node's storage—this method is available only in **DEBUG** mode for unit testing.

Code Listing 57: SimpleStore

```
#if DEBUG           // For unit testing
public void SimpleStore(ID key, string val)
{
  storage.Set(key, val);
}
#endif
```

ValueFoundInFartherNode

We can change the setup of the IDs and verify that the we find the value in a farther node.

Code Listing 58: ValueStoredInFartherNodeTest

```
[TestMethod]
public void ValueStoredInFartherNodeTest()
{
  VirtualProtocol vp1 = new VirtualProtocol();
  VirtualProtocol vp2 = new VirtualProtocol();
  VirtualStorage store1 = new VirtualStorage();
  VirtualStorage store2 = new VirtualStorage();

  // Ensures that all nodes are closer, because ID.Max ^ n < ID.Max when
n > 0.
  Dht dht = new Dht(ID.Zero, vp1, new Router(), store1, store1,
    new VirtualStorage());
  vp1.Node = dht.Router.Node;

  ID contactID = ID.Max;      // a farther contact.
  Contact otherContact = new Contact(vp2, contactID);
  Node otherNode = new Node(otherContact, store2);
  vp2.Node = otherNode;

  // Add this other contact to our peer list.
  dht.Router.Node.BucketList.AddContact(otherContact);

  // We want an integer distance, not an XOR distance.
  ID key = ID.One;
```

```
  // Set the value in the other node, to be discovered by the lookup
process.
  string val = "Test";
  otherNode.SimpleStore(key, val);


  Assert.IsFalse(store1.Contains(key),
    "Expected our peer to NOT have cached the key-value.");
  Assert.IsTrue(store2.Contains(key),
    "Expected other node to HAVE cached the key-value.");


  // Try and find the value, given our Dht knows about the other contact.
  string retval = dht.FindValue(key).val;


  Assert.IsTrue(retval == val, "Expected to get back what we stored");
}
```

ValueStoredGetsPropagated

Here we test that when we store a value to our peer, it also gets propagated to another peer that our peer knows about.

Code Listing 59: ValueStoredGetsPropagatedTest

```
[TestMethod]
public void ValueStoredGetsPropagatedTest()
{
  VirtualProtocol vp1 = new VirtualProtocol();
  VirtualProtocol vp2 = new VirtualProtocol();
  VirtualStorage store1 = new VirtualStorage();
  VirtualStorage store2 = new VirtualStorage();
```

```
  // Ensures that all nodes are closer, because ID.Max ^ n < ID.Max when
n > 0.
  Dht dht = new Dht(ID.Max, vp1, new Router(), store1, store1, new
VirtualStorage());
  vp1.Node = dht.Router.Node;


  ID contactID = ID.Mid;      // a closer contact.
  Contact otherContact = new Contact(vp2, contactID);
  Node otherNode = new Node(otherContact, store2);
  vp2.Node = otherNode;


  // Add this other contact to our peer list.
  dht.Router.Node.BucketList.AddContact(otherContact);


  // We want an integer distance, not an XOR distance.
  ID key = ID.Zero;
  string val = "Test";


  Assert.IsFalse(store1.Contains(key), "Obviously we don't have the key-
value yet.");
  Assert.IsFalse(store2.Contains(key),
    "And equally obvious, the other peer doesn't have the key-value yet
either.");


  dht.Store(key, val);


  Assert.IsTrue(store1.Contains(key),
    "Expected our peer to have stored the key-value.");
  Assert.IsTrue(store2.Contains(key),
    "Expected the other peer to have stored the key-value.");
}
```

GetValuePropagatesToCloserNode

This test verifies that, given three nodes (the first of which is us), where node 2 has the value, a get value also propagates to node 3 because a lookup was performed.

Code Listing 60: GetValuePropagatesToCloserNodeTest

```
[TestMethod]
public void GetValuePropagatesToCloserNodeTest()
{
  VirtualProtocol vp1 = new VirtualProtocol();
  VirtualProtocol vp2 = new VirtualProtocol();
  VirtualProtocol vp3 = new VirtualProtocol();
  VirtualStorage store1 = new VirtualStorage();
  VirtualStorage store2 = new VirtualStorage();
  VirtualStorage store3 = new VirtualStorage();
  VirtualStorage cache3 = new VirtualStorage();

  // Ensures that all nodes are closer, because ID.Max ^ n < ID.Max when
n > 0.
  Dht dht = new Dht(ID.Max, vp1, new Router(), store1, store1, new
VirtualStorage());
  vp1.Node = dht.Router.Node;

  // Setup node 2:

  ID contactID2 = ID.Mid;      // a closer contact.
  Contact otherContact2 = new Contact(vp2, contactID2);
  Node otherNode2 = new Node(otherContact2, store2);
  vp2.Node = otherNode2;

  // Add the second contact to our peer list.
```

```
  dht.Router.Node.BucketList.AddContact(otherContact2);


  // Node 2 has the value.
  // We want an integer distance, not an XOR distance.
  ID key = ID.Zero;
  string val = "Test";
  otherNode2.Storage.Set(key, val);


  // Setup node 3:


  ID contactID3 = ID.Zero.SetBit(158);      // 01000.... -- a farther
contact.
  Contact otherContact3 = new Contact(vp3, contactID3);
  Node otherNode3 = new Node(otherContact3, store3, cache3);
  vp3.Node = otherNode3;


  // Add the third contact to our peer list.
  dht.Router.Node.BucketList.AddContact(otherContact3);


  Assert.IsFalse(store1.Contains(key), "Obviously we don't have the key-
value yet.");
  Assert.IsFalse(store3.Contains(key),
    "And equally obvious, the third peer doesn't have the key-value yet
either.");


  var ret = dht.FindValue(key);


  Assert.IsTrue(ret.found, "Expected value to be found.");
  Assert.IsFalse(store3.Contains(key), "Key should not be in the
republish store.");
```

```
  Assert.IsTrue(cache3.Contains(key), "Key should be in the cache
store.");

  Assert.IsTrue(cache3.GetExpirationTimeSec(key.Value) ==
    Constants.EXPIRATION_TIME_SECONDS / 2, "Expected 12 hour
expiration.");

}
```

Chapter 8 The Dht–Bootstrapping

From the spec: "*To join the network, a node u must have a contact to an already participating node w. u inserts w into the appropriate k-bucket. u then performs a node lookup for its own node ID. Finally, u refreshes all k-buckets further away than its closest neighbor. During the refreshes, u both populates its own k-buckets and inserts itself into other nodes' k-buckets as necessary.*"

The Wikipedia page adds a little more detail:

"The joining node inserts the bootstrap node into one of its k-buckets. The joining node then does a FIND_NODE of its own ID against the bootstrap node (the only other node it knows). The "self-lookup" will populate other nodes' k-buckets with the new node ID, and will populate the joining node's k-buckets with the nodes in the path between it and the bootstrap node. After this, the joining node refreshes all k-buckets further away than the k-bucket the bootstrap node falls in. This refresh is just a lookup of a random key that is within that k-bucket range."

By choosing a random ID within the contact's bucket range, we are creating an ID whose prefix determines the ordering of the contacts returned by **`GetCloseContacts`**:

```
Select(c => new { contact = c, distance = c.ID.Value ^ key.Value }).OrderBy(d => d.distance)
```

This will sort the contacts such that those that are closer—those where no bits are set in the prefix of the contact—are first in the list. Ideally, with many peers participating, we should get **`k`** contacts that are closer.

Of particular note here is that when a peer network is small or in the throes of being born, other contacts that nodes have will not be discovered until the bootstrapping bucket splits. We'll see how the network self-corrects later on. It's also interesting to realize that "joining" actually means contacting another node with any one of the four RPC calls. A new peer could join an existing network with its first RPC being FindValue!

Bootstrapping implementation

Getting a random **`ID`** within a bucket range is based on knowing that bucket ranges are always powers of 2. We use this for unit testing.

Code Listing 61: RandomIDWithinBucket

```
/// <summary>
/// Returns an ID within the range of the bucket's Low and High range.
/// The optional parameter forceBit1 is for our unit tests.
```

```
/// This works because the bucket low-high range will always be a power
of 2!
/// </summary>
public static ID RandomIDWithinBucket(KBucket bucket, bool forceBit1 =
false)
{
  // Simple case:
  // High = 1000
  // Low  = 0010
  // We want random values between 0010 and 1000

  // Low and High will always be powers of 2.
  var lowBits = new ID(bucket.Low).Bytes.Bits().Reverse();
  var highBits = new ID(bucket.High).Bytes.Bits().Reverse();

  // We randomize "below" this High prefix range.
  int highPrefix = highBits.TakeWhile(b => !b).Count() + 1;
  // Up to the prefix of the Low range.
  // This sets up a mask of 0's for the LSB's in the Low prefix.
  int lowPrefix = lowBits.TakeWhile(b => !b).Count();
  // RandomizeBeyond is little endian for "bits after" so reverse
high/low prefixes.
  ID id = Zero.RandomizeBeyond(Constants.ID_LENGTH_BITS - highPrefix,
    Constants.ID_LENGTH_BITS - lowPrefix, forceBit1);

  // The we add the low range.
  id = new ID(bucket.Low + id.Value);

  return id;
}
```

Bootstrapping

The actual bootstrap implementation is straightforward.

Code Listing 62: DHT Bootstrap

```
/// <summary>
/// Bootstrap our peer by contacting another peer, adding its contacts
/// to our list, then getting the contacts for other peers not in the
/// bucket range of our known peer we're joining.
/// </summary>
public RpcError Bootstrap(Contact knownPeer)
{
  node.BucketList.AddContact(knownPeer);
  var (contacts, error) = knownPeer.Protocol.FindNode(ourContact, ourId);
  HandleError(error, knownPeer);

  if (!error.HasError)
  {
    contacts.ForEach(c => node.BucketList.AddContact(c));
    KBucket knownPeerBucket = node.BucketList.GetKBucket(knownPeer.ID);
    // Resolve the list now, so we don't include additional contacts as
we
    // add to our bucket additional contacts.
    var otherBuckets = node.BucketList.Buckets.Where(
      b => b != knownPeerBucket).ToList();
    otherBuckets.ForEach(b => RefreshBucket(b));

    foreach (KBucket otherBucket in otherBuckets)
    {
      RefreshBucket(otherBucket);
    }
```

```
  }


  return error;
}

protected void RefreshBucket(KBucket bucket)
{
  bucket.Touch();
  ID rndId = ID.RandomIDWithinBucket(bucket);
  // Isolate in a separate list as contacts collection for this bucket
might change.
  List<Contact> contacts = bucket.Contacts.ToList();


  contacts.ForEach(c =>
  {
    var (newContacts, timeoutError) = c.Protocol.FindNode(ourContact,
rndId);
    HandleError(timeoutError, c);
    newContacts?.ForEach(otherContact =>
node.BucketList.AddContact(otherContact));
  });
}
```

Bootstrapping unit tests

Getting a random ID within a bucket range is complicated enough that it deserves a unit test.

Code Listing 63: RandomWithinBucketTests

```
[TestMethod]
public void RandomWithinBucketTests()
{
```

```
  // Must be powers of 2.
  List<(int low, int high)> testCases = new List<(int low, int high)>()
  {
    (0, 256),              // 7 bits should be set
     (256, 1024),          // 2 bits (256 + 512) should be set
     (65536, 65536 * 2),   // no additional bits should be set.
     (65536, 65536 * 4),   // 2 bits (65536 and 65536*2) should be set.
     (65536, 65536 * 16),  // 4 bits (65536, 65536*2, 65536*4, 65536*8)
set.
  };

  foreach (var testCase in testCases)
  {
    KBucket bucket = new KBucket(testCase.low, testCase.high);
    // We force all bits in the range we are "randomizing" to be true
    // so it's not really randomized.  This verifies the outer algorithm
    // that figures out which bits to randomize.
    ID id = ID.RandomIDWithinBucket(bucket, true);

    Assert.IsTrue(id >= bucket.Low && id < bucket.High,
      "ID is outside of bucket range.");

    // The ID, because we're forcing bits, should always be (high - 1)
and
    // ~max(0, low - 1)
    int bitCheck = (testCase.high - 1) & ~Math.Max(0, testCase.low - 1);

    Assert.IsTrue(id.Value == bitCheck, "Expected bits are not
correct.");
  }
}
```

BootstrapWithinBootstrappingBucket

In the actual bootstrapping unit test, we are setting up a bootstrapping peer we are joining to with 10 contacts. One of those contacts also knows about 10 other contacts. The joining peer will receive 10 contacts (for a total of 11, the bootstrapper + 10) and will not find any others because the "other peers not in the known peer bucket" are all in the same bucket (the bucket hasn't split yet). The IDs for our peers are irrelevant in this scenario.

Code Listing 64: BootstrapWithinBootstrappingBucketTest

```
[TestMethod]
public void BootstrapWithinBootstrappingBucketTest()
{
  // We need 22 virtual protocols.  One for the bootstrap peer,
  // 10 for the nodes the bootstrap peer knows about, and 10 for the
nodes
  // one of those nodes knows about, and one for us to rule them all.
  VirtualProtocol[] vp = new VirtualProtocol[22];
  22.ForEach((i) => vp[i] = new VirtualProtocol());


  // Us
  Dht dhtUs = new Dht(ID.RandomID, vp[0], () => new VirtualStorage(), new
Router());
  vp[0].Node = dhtUs.Router.Node;


  // Our bootstrap peer
  Dht dhtBootstrap = new Dht(ID.RandomID, vp[1],
    () => new VirtualStorage(), new Router());
  vp[1].Node = dhtBootstrap.Router.Node;
  Node n = null;


  // Our boostrapper knows 10 contacts
  10.ForEach((i) =>
  {
```

```
    Contact c = new Contact(vp[i + 2], ID.RandomID);
    n = new Node(c, new VirtualStorage());
    vp[i + 2].Node = n;
    dhtBootstrap.Router.Node.BucketList.AddContact(c);
  });

  // One of those nodes, in this case the last one we added to our
bootstrapper
  // for convenience, knows about 10 other contacts.
  10.ForEach((i) =>
  {
    Contact c = new Contact(vp[i + 12], ID.RandomID);
    Node n2 = new Node(c, new VirtualStorage());
    vp[i + 12].Node = n;
    n.BucketList.AddContact(c); // Note we're adding these contacts to
the 10th node.
  });

  dhtUs.Bootstrap(dhtBootstrap.Router.Node.OurContact);

  Assert.IsTrue(dhtUs.Router.Node.BucketList.Buckets.Sum(
    c => c.Contacts.Count) == 11, "Expected our peer to get 11
contacts.");
}
```

BootstrapOutsideBootstrappingBucket

In this test, we set up 20 nodes in the bootstrap peer so that we know how the buckets split *for us* (20 in the left one, one in the right one) and add 10 contacts to the one in the right one. Because out bootstrap peer will be in our left bucket, we should have a total of 31 contacts (bootstrap + its 20 contacts + the other nodes 10 contacts).

Code Listing 65: BootstrapOutsideBootstrappingBucketTest

```
[TestMethod]
public void BootstrapOutsideBootstrappingBucketTest()
{
  // We need 32 virtual protocols. One for the bootstrap peer,
  // 20 for the nodes the bootstrap peer knows about, 10 for the nodes
  // one of those nodes knows about, and one for us to rule them all.
  VirtualProtocol[] vp = new VirtualProtocol[32];
  32.ForEach((i) => vp[i] = new VirtualProtocol());

  // Us, ID doesn't matter.
  Dht dhtUs = new Dht(ID.RandomID, vp[0], () => new VirtualStorage(), new
Router());
  vp[0].Node = dhtUs.Router.Node;

  // Our bootstrap peer
  // All IDs are < 2^159
  Dht dhtBootstrap = new
Dht(ID.Zero.RandomizeBeyond(Constants.ID_LENGTH_BITS - 1),
    vp[1], () => new VirtualStorage(), new Router());
  vp[1].Node = dhtBootstrap.Router.Node;
  Node n = null;

  // Our boostrapper knows 20 contacts
  20.ForEach((i) =>
  {
    ID id;

    // All IDs are < 2^159 except the last one, which is >= 2^159
```

```
    // which will force a bucket split for _us_
    if (i < 19)
    {
      id = ID.Zero.RandomizeBeyond(Constants.ID_LENGTH_BITS - 1);
    }
    else
    {
      id = ID.Max;
    }

    Contact c = new Contact(vp[i + 2], id);
    n = new Node(c, new VirtualStorage());
    vp[i + 2].Node = n;
    dhtBootstrap.Router.Node.BucketList.AddContact(c);
  });

  // One of those nodes, in this case specifically the last one we added
to
  // our bootstrapper so that it isn't in the bucket of our bootstrapper,
  // we add 10 contacts.  The IDs of those contacts don't matter.
  10.ForEach((i) =>
  {
    Contact c = new Contact(vp[i + 22], ID.RandomID);
    Node n2 = new Node(c, new VirtualStorage());
    vp[i + 22].Node = n;
    n.BucketList.AddContact(c);// Note we're adding these contacts to the
10th node.
  });

  dhtUs.Bootstrap(dhtBootstrap.Router.Node.OurContact);
```

```
  Assert.IsTrue(dhtUs.Router.Node.BucketList.Buckets.Sum(
    c => c.Contacts.Count) == 31, "Expected our peer to have 31
contacts.");

}
```

Chapter 9 Bucket Management

It is time now to deal with the various issues of bucket management.

Eviction and queuing new contacts

When the bucket is full, we address evicting unresponsive contacts in full buckets as well as queuing pending contacts. We simulate this for unit testing with our **`VirtualProtocol`** in the **`AddContact`** method by setting up a flag that is used to indicate that the node did not respond.

Code Listing 66: VirtualProtocol Ping

```
public RpcError Ping(Contact sender)
{
  // Ping still adds/updates the sender's contact.
  if (Responds)
  {
    Node.Ping(sender);
  }

  return new RpcError() { TimeoutError = !Responds };
}
```

We can then implement the handling of a nonresponsive node.

Code Listing 67: Handling Nonresponsive Nodes (fragment)

```
else if (kbucket.IsBucketFull)
{
  if (CanSplit(kbucket))
  {
    // Split the bucket and try again.
    (KBucket k1, KBucket k2) = kbucket.Split();
```

```
    int idx = GetKBucketIndex(contact.ID);
    buckets[idx] = k1;
    buckets.Insert(idx + 1, k2);
    buckets[idx].Touch();
    buckets[idx + 1].Touch();
    AddContact(contact);
  }
  else
  {
    Contact lastSeenContact = kbucket.Contacts.OrderBy(c =>
c.LastSeen).First();
    RpcError error = lastSeenContact.Protocol.Ping(ourContact);

    if (error.HasError)
    {
      // Null continuation is used because unit tests may not initialize
a DHT.
      dht?.DelayEviction(lastSeenContact, contact);
    }
    else
    {
      // Still can't add the contact, so put it into the pending list.
      dht?.AddToPending(contact);
    }
  }
}
```

Bucket refresh

From the spec: "*Buckets are generally kept fresh by the traffic of requests traveling through nodes. To handle pathological cases in which there are no lookups for a particular ID range, each node refreshes any bucket to which it has not performed a node lookup in the past hour. Refreshing means picking a random ID in the bucket's range and performing a node search for that ID.*"

The phrase "any bucket to which it has not performed a node lookup" is subject to at least two interpretations. One way to interpret this is possibly "the bucket whose range contains the key in the key-value pair for a **`Store`** or **`FindValue`** operation. Another interpretation is "the k-bucket containing the range for any contact ID queried during during the lookup process." This second approach might seem more correct because the original alpha contacts is determined from the list of closest contacts across all buckets, but it then becomes arbitrary as to whether to also touch the buckets containing the contacts returned by the **`FindNodes`** query that are then queried further.

I am choosing the first interpretation, which means that the bucket containing the key gets touched in the **`Store`** and **`FindValue`** methods of the **`Dht`** class.

Code Listing 68: TouchBucketWithKey

```
public class Dht
{
  ...
  public void Store(ID key, string val)
  {
    TouchBucketWithKey(key);
    ...
  }

  public (bool found, List<Contact> contacts, string val) FindValue(ID
key)
  {
    TouchBucketWithKey(key);
    ...
  }

  protected void TouchBucketWithKey(ID key)
  {
    node.BucketList.GetKBucket(key).Touch();
```

```
  }

}
```

Next, we set up a refresh timer in the **Dht** class.

Code Listing 69: SetupBucketRefreshTimer

```
protected void SetupBucketRefreshTimer()
{
  bucketRefreshTimer = new Timer(Constants.BUCKET_REFRESH_INTERVAL);
  bucketRefreshTimer.AutoReset = true;
  bucketRefreshTimer.Elapsed += BucketRefreshTimerElapsed;
  bucketRefreshTimer.Start();
}

protected void BucketRefreshTimerElapsed(object sender, ElapsedEventArgs
e)
{
  DateTime now = DateTime.Now;


  // Put into a separate list as bucket collections may be modified.
  List<KBucket> currentBuckets =
    new List<KBucket>(node.BucketList.Buckets.
    Where(b => (now - b.TimeStamp).TotalMilliseconds >=
      Constants.BUCKET_REFRESH_INTERVAL));


  currentBuckets.ForEach(b => RefreshBucket(b));
}

protected void RefreshBucket(KBucket bucket)
{
  bucket.Touch();
  ID rndId = ID.RandomIDWithinBucket(bucket);
```

```
  // Isolate in a separate list as contacts collection for this bucket
might change.
  List<Contact> contacts = bucket.Contacts.ToList();


  contacts.ForEach(c =>
  {
    var (newContacts, timeoutError) = c.Protocol.FindNode(ourContact,
rndId);
    HandleError(timeoutError, c);
    newContacts?.ForEach(otherContact =>
node.BucketList.AddContact(otherContact));
  });
}
```

Note that now when a bucket is refreshed, it is always touched, which updates its “last seen” timestamp.

Unit tests

Two unit tests for eviction verify the two possible conditions.

- A nonresponding contact is evicted after a present number of ping attempts.
- A new contact is placed into a delayed eviction buffer.

Code Listing 70: NonRespondingContactEvictedTest

```
/// <summary>
/// Tests that a nonresponding contact is evicted after
/// Constant.EVICTION_LIMIT tries.
/// </summary>
[TestMethod]
public void NonRespondingContactEvictedTest()
{
  // Create a DHT so we have an eviction handler.
```

```
  Dht dht = new Dht(ID.Zero, new VirtualProtocol(), () => null, new
Router());


  IBucketList bucketList = SetupSplitFailure(dht.Node.BucketList);


  Assert.IsTrue(bucketList.Buckets.Count == 2, "Bucket split should have
occurred.");
  Assert.IsTrue(bucketList.Buckets[0].Contacts.Count == 1,
    "Expected 1 contact in bucket 0.");
  Assert.IsTrue(bucketList.Buckets[1].Contacts.Count == 20,
    "Expected 20 contacts in bucket 1.");


  // The bucket is now full. Pick the first contact, as it is last
 // seen (they are added in chronological order.)
  Contact nonRespondingContact = bucketList.Buckets[1].Contacts[0];


  // Since the protocols are shared, we need to assign a unique protocol
  // for this node for testing.
  VirtualProtocol vpUnresponding =
    new
VirtualProtocol(((VirtualProtocol)nonRespondingContact.Protocol).Node,
      false);
  nonRespondingContact.Protocol = vpUnresponding;


  // Setup the next new contact (it can respond.)
  Contact nextNewContact = new Contact(dht.Contact.Protocol,
ID.Zero.SetBit(159));


  // Hit the nonresponding contact EVICTION_LIMIT times, which will
trigger
  // the eviction algorithm.
  Constants.EVICTION_LIMIT.ForEach(() =>
bucketList.AddContact(nextNewContact));
```

```
  Assert.IsTrue(bucketList.Buckets[1].Contacts.Count == 20,
    "Expected 20 contacts in bucket 1.");


  // Verify CanSplit -> Pending eviction happened.


  Assert.IsTrue(dht.PendingContacts.Count == 0,
    "Pending contact list should now be empty.");
Assert.IsFalse(bucketList.Buckets.SelectMany(
    b => b.Contacts).Contains(nonRespondingContact),
      "Expected bucket to NOT contain non-responding contact.");
Assert.IsTrue(bucketList.Buckets.SelectMany(
    b => b.Contacts).Contains(nextNewContact),
      "Expected bucket to contain new contact.");
  Assert.IsTrue(dht.EvictionCount.Count == 0,
    "Expected no contacts to be pending eviction.");
}
```

Code Listing 71: NonRespondingContactDelayedEvictionTest

```
/// <summary>
/// Tests that a nonresponding contact puts the new contact into a
pending list.
/// </summary>
[TestMethod]
public void NonRespondingContactDelayedEvictionTest()
{
  // Create a DHT so we have an eviction handler.
  Dht dht = new Dht(ID.Zero, new VirtualProtocol(), () => null, new
Router());


  IBucketList bucketList = SetupSplitFailure(dht.Node.BucketList);
```

```
  Assert.IsTrue(bucketList.Buckets.Count == 2,
    "Bucket split should have occurred.");

  Assert.IsTrue(bucketList.Buckets[0].Contacts.Count == 1,
    "Expected 1 contact in bucket 0.");

  Assert.IsTrue(bucketList.Buckets[1].Contacts.Count == 20,
    "Expected 20 contacts in bucket 1.");



  // The bucket is now full. Pick the first contact, as it is
  // last seen (they are added in chronological order.)

  Contact nonRespondingContact = bucketList.Buckets[1].Contacts[0];


  // Since the protocols are shared, we need to assign a unique protocol
  // for this node for testing.

  VirtualProtocol vpUnresponding = new

VirtualProtocol(((VirtualProtocol)nonRespondingContact.Protocol).Node,
false);

  nonRespondingContact.Protocol = vpUnresponding;


  // Setup the next new contact (it can respond.)

  Contact nextNewContact = new Contact(dht.Contact.Protocol,
ID.Zero.SetBit(159));


  bucketList.AddContact(nextNewContact);


  Assert.IsTrue(bucketList.Buckets[1].Contacts.Count == 20,
    "Expected 20 contacts in bucket 1.");


  // Verify CanSplit -> Evict happened.
```

```
  Assert.IsTrue(dht.PendingContacts.Count == 1, "Expected one pending
contact.");

  Assert.IsTrue(dht.PendingContacts.Contains(nextNewContact),
    "Expected pending contact to be the 21st contact.");

  Assert.IsTrue(dht.EvictionCount.Count == 1,
    "Expected one contact to be pending eviction.");

}
```

Both unit tests setup a “failed split” so that the eviction routines are triggered.

Code Listing 72: SetupSplitFailure

```
protected IBucketList SetupSplitFailure(IBucketList bucketList = null)

{

  // force host node ID to < 2^159 so the node ID is not in the 2^159 ...
2^160 range

  byte[] bhostID = new byte[20];

  bhostID[19] = 0x7F;

  ID hostID = new ID(bhostID);


  Contact dummyContact = new Contact(new VirtualProtocol(), hostID);

   ((VirtualProtocol)dummyContact.Protocol).Node =
      new Node(dummyContact, new VirtualStorage());

  bucketList = bucketList ?? new BucketList(hostID, dummyContact);


  // Also add a contact in this 0 - 2^159 range, arbitrarily something
  // not our host ID. This ensures that only one bucket split will occur
  // after 20 nodes with ID >= 2^159 are added,

  // otherwise, buckets will in the 2^159 ... 2^160 space.

  dummyContact = new Contact(new VirtualProtocol(), ID.One);

   ((VirtualProtocol)dummyContact.Protocol).Node =
    new Node(dummyContact, new VirtualStorage());

    bucketList.AddContact(new Contact(dummyContact.Protocol, ID.One));
```

```
  Assert.IsTrue(bucketList.Buckets.Count == 1,
    "Bucket split should not have occurred.");

  Assert.IsTrue(bucketList.Buckets[0].Contacts.Count == 1,
    "Expected 1 contact in bucket 0.");


  // make sure contact IDs all have the same 5-bit prefix
  // and are in the 2^159 ... 2^160 - 1 space

  byte[] bcontactID = new byte[20];

  bcontactID[19] = 0x80;

  // 1000 xxxx prefix, xxxx starts at 1000 (8)

  // this ensures that all the contacts in a bucket match only the prefix
as
  // only the first 5 bits are shared.

  // |----| shared range

  // 1000 1000 ...

  // 1000 1100 ...

  // 1000 1110 ...

  byte shifter = 0x08;

  int pos = 19;


  Constants.K.ForEach(() =>

  {

    bcontactID[pos] |= shifter;

    ID contactID = new ID(bcontactID);

    dummyContact = new Contact(new VirtualProtocol(), ID.One);

    ((VirtualProtocol)dummyContact.Protocol).Node =
       new Node(dummyContact, new     VirtualStorage());

    bucketList.AddContact(new Contact(dummyContact.Protocol, contactID));

    shifter >>= 1;
```

```
    if (shifter == 0)
    {
      shifter = 0x80;
      --pos;
    }
  });

  return bucketList;
}
```

Chapter 10 Key-Value Management

In this section, you will learn something that is not at all clearly stated in the Kademlia specification—there are actually three kinds of data store:

- Original publisher store
- Republished store
- Cached store

It is important to know that the Kademlia specification does not discuss this at all. The information in this section has been gleaned from closely looking at the discussion on the eMule project forum, particularly **miniminime**'s discussion of the roles that a node can take on.[14] Implementing this approach requires that the receiver peer knows whether to store the key-value in the republish store or in the cache store.

Original publisher store

The original publisher store is never written to as a result of an RPC store. This store is updated only by the peer itself when it is the originator of the key-value and wants to publish the key-value to other peers. Key-values from this store are republished once every 24 hours.

Republished store

This store contains key-values that have been republished by other peers as part of process of distributing the data among peers. This store never contains the peer's (as an originator) own storage, but only key-values received from other peers. Key-values in this store are republished to other peers only if a closer peer is found. This check occurs every hour and is optimized to avoid calling the lookup algorithm:

- For a particular bucket if it has already been queried for closer nodes.
- If the key has been republished in the last hour.

Cached store

The cached store is used when republishing a **`FindValue`** request onto the *next closest node*. The intention here is to avoid hotspots during lookup of popular keys by temporarily republishing them onto other nearby peers. These temporary key-values have an expiration time. Key-values in the cached store are never republished, and are removed after the expiration time.

[14] https://forum.emule-project.net/index.php?showtopic=32335&view=findpost&p=214837

Storage mechanisms in the Dht class

The three storage mechanisms are managed in the **Dht** class.

Code Listing 73: Storage Mechanisms

```
protected IStorage originatorStorage;

protected IStorage republishStorage;

protected IStorage cacheStorage;
```

The **Dht** class implements a few constructor options, the first primarily for aiding unit testing.

Code Listing 74: Dht Constructors

```
/// <summary>

/// Use this constructor to initialize the stores to the same instance.

/// </summary>

public Dht(
  ID id,
  IProtocol protocol,
  Func<IStorage> storageFactory, BaseRouter router)

{

  originatorStorage = storageFactory();

  republishStorage = storageFactory();

  cacheStorage = storageFactory();

  FinishInitialization(id, protocol, router);

  SetupTimers();

}


/// <summary>

/// Supports different concrete storage types. For example, you may want
/// the cacheStorage to be an in-memory store, the originatorStorage to
be
/// a SQL database, and the republish store to be a key-value database.

/// </summary>
```

```
public Dht(
  ID id,
  IProtocol protocol,
  BaseRouter router,
  IStorage originatorStorage,
  IStorage republishStorage,
  IStorage cacheStorage)

{

  this.originatorStorage = originatorStorage;

  this.republishStorage = republishStorage;

  this.cacheStorage = cacheStorage;

  FinishInitialization(id, protocol, router);

  SetupTimers();

}
```

The ability to specify different storage mechanisms can be very useful; however, this means that a **Node** must store the key-value in the appropriate storage.

Code Listing 75: The Node Store Method

```
/// <summary>

/// Store a key-value pair in the republish or cache storage.

/// </summary>

public void Store(
  Contact sender,
  ID key,
  string val,
  bool isCached = false,
  int expirationTimeSec = 0)

{

  Validate.IsFalse<SendingQueryToSelfException>(sender.ID ==
ourContact.ID,
    "Sender should not be ourself!");

  bucketList.AddContact(sender);
```

```
  if (isCached)
  {
    cacheStorage.Set(key, val, expirationTimeSec);
  }
  else
  {
    SendKeyValuesIfNewContact(sender);
    storage.Set(key, val, Constants.EXPIRATION_TIME_SECONDS);
  }
}
```

Republishing key-values

From the spec: "*To ensure the persistence of key-value pairs, nodes must periodically republish keys. Otherwise, two phenomena may cause lookups for valid keys to fail. First, some of the k nodes that initially get a key-value pair when it is published may leave the network. Second, new nodes may join the network with IDs closer to some published key than the nodes on which the key-value pair was originally published. In both cases, the nodes with a key-value pair must republish it so as once again to ensure it is available on the k nodes closest to the key.*

To compensate for nodes leaving the network, Kademlia republishes each key-value pair once an hour. A naive implementation of this strategy would require many messages—each of up to k nodes storing a key-value pair would perform a node lookup followed by k - 1 STORE RPCs every hour."

From Wikipedia,[15] which can be helpful for understanding the spec with different phrasing:

"Periodically, a node that stores a value will explore the network to find the k nodes that are close to the key value and replicate the value onto them. This compensates for disappeared nodes."

And:

"The node that is providing the file [key-value] will periodically refresh the information onto the network (perform FIND_NODE and STORE messages). When all of the nodes having the file [key-value] go offline, nobody will be refreshing its values (sources and keywords) and the information will eventually disappear from the network."

[15] https://en.wikipedia.org/wiki/Kademlia

The Wikipedia write-up clarifies what is meant by “on the k nodes closest to the key.” In other words, for each key, a **`FindNode`** is called to find closer nodes, and the value is republished. Without the optimizations, this can be a time-consuming process if there’s a lot of key-values in a node’s store, which is addressed in an optimization later.

First optimization

From the spec: “*Fortunately, the republishing process can be heavily optimized. First, when a node receives a STORE RPC for a given key-value pair, it assumes the RPC was also issued to the other k - 1 closest nodes, and thus the recipient will not republish the key-value pair in the next hour. This ensures that as long as republication intervals are not exactly synchronized, only one node will republish a given key-value pair every hour.*”

This first optimization is simple—when receiving a store, update the timestamp on the key-value. Any key-value that has been touched within the last hour is not republished, as we can assume:

- For a new key-value, it was also published to *k* closer nodes.
- If it’s been republished by another node, that node republished it to *k* closer nodes.

Second optimization

From the spec: “*A second optimization avoids performing node lookups before republishing keys. As described in Section 2.4, to handle unbalanced trees, nodes split k-buckets as required to ensure they have complete knowledge of a surrounding subtree with at least k nodes. If, before republishing key-value pairs, a node u refreshes all k-buckets in this subtree of k nodes, it will automatically be able to figure out the k closest nodes to a given key. These bucket refreshes can be amortized over the republication of many keys.*”

This second optimization is straightforward—if we’ve done a bucket refresh within the last hour, we can avoid calling **`FindNode`** (the node lookup algorithm.) How do we determine the bucket to test if it’s been refreshed? The bucket for which the key is in range should contain some closer contacts we’ve seen for that key. While the answer might be obvious, it’s worthwhile to discuss the reasoning here.

Buckets in the bucket list are maintained in range order rather than in a tree, which naturally orders them by their prefix.

Table 1: Bucket Range(s)

State	Bucket Range(s)	Prefix(es)
Initial Bucket	$0 .. 2^{160}$	1
Two Buckets	$0 .. 2^{159} \mid 2^{159} .. 2^{160}$	01, 1

State	Bucket Range(s)	Prefix(es)
Four Buckets	$0 .. 2^{158} \mid 2^{158} .. 2^{159} \mid 2^{159} - 2^{159} + 2^{158} \mid 2^{159} - 2^{159}+2^{158} .. 2^{160}$	001, 01, 10, 1

When we identify a bucket with a given key, the contacts in that bucket are closest, as per the XOR computation on the prefix. For example, looking at the four buckets with prefixes 001, 01, 10, and 1, we see that the contacts in the key's bucket range are closest (the closest bucket contacts are in green, and farther bucket contacts are in red).

Table 2: Key Prefixes

Key Prefix	Key Prefix ^ Bucket Prefixes	Explanation
1	**101, 11, 01, 0**	Bucket with prefix 1 always has contacts that are closer
01	**011, 00, 11, 11**	Bucket with prefix 01 always has contacts that are closer
001	**000, 011, 101, 101**	Bucket with prefix 001 always has contacts that are closer
0001	**0011, 0101, 1001, 1001**	Bucket with prefix 001 always has contacts that are closer

For this reason, we use the bucket for which the key is in range. Also, new key-values that are published onto to closer nodes persist for 24 hours.

Implementation

Key-values in the republish store are republished at a particular interval, typically every hour.

Code Listing 76: KeyValueRepublishElapsed

```
/// <summary>
/// Replicate key values if the key-value hasn't been touched within
/// the republish interval. Also don't do a FindNode lookup if the bucket
/// containing the key has been refreshed within the refresh interval.
/// </summary>
protected void KeyValueRepublishElapsed(object sender, ElapsedEventArgs
e)
{
```

```
  DateTime now = DateTime.Now;


  republishStorage.Keys.Where(k =>
    (now - republishStorage.GetTimeStamp(k)).TotalMilliseconds >=
      Constants.KEY_VALUE_REPUBLISH_INTERVAL).ForEach(k=>
  {
    ID key = new ID(k);
    StoreOnCloserContacts(key, republishStorage.Get(key));
    republishStorage.Touch(k);
  });
}
```

Note how a lookup is only performed if the bucket containing the key hasn't itself been refreshed recently (within the past hour).

Code Listing 77: StoreOnCloserContacts

```
/// <summary>
/// Perform a lookup if the bucket containing the key has not been
refreshed,
/// otherwise just get the contacts the k closest contacts we know about.
/// </summary>
protected void StoreOnCloserContacts(ID key, string val)
{
  DateTime now = DateTime.Now;


  KBucket kbucket = node.BucketList.GetKBucket(key);
  List<Contact> contacts;


  if ((now - kbucket.TimeStamp).TotalMilliseconds <
    Constants.BUCKET_REFRESH_INTERVAL)
  {
```

```
    // Bucket has been refreshed recently, so don't do a lookup as we
    // have the k closes contacts.
    contacts = node.BucketList.GetCloseContacts(key, node.OurContact.ID);
  }
  else
  {
    contacts = router.Lookup(key, router.RpcFindNodes).contacts;
  }

  contacts.ForEach(c =>
  {
    RpcError error = c.Protocol.Store(node.OurContact, key, val);
    HandleError(error, c);
  });
}
```

Expiring key-value pairs

Expired key-values are removed from the republish and cache storage, which happens in the **`Dht`** class.

Code Listing 78: ExpireKeysElapsed

```
/// <summary>
/// Any expired keys in the republish or node's cache are removed.
/// </summary>
protected virtual void ExpireKeysElapsed(object sender, ElapsedEventArgs
e)
{
  RemoveExpiredData(cacheStorage);
  RemoveExpiredData(republishStorage);
```

```
}


protected void RemoveExpiredData(IStorage store)
{
  DateTime now = DateTime.Now;
  // ToList so our key list is resolved now as we remove keys.
  store.Keys.Where(k => (now - store.GetTimeStamp(k)).TotalSeconds >=
    store.GetExpirationTimeSec(k)).ToList().ForEach(k =>
  {
    store.Remove(k);
  });
}
```

Originator republishing

From the spec: "*For Kademlia's current application (file sharing), we also require the original publisher of a (key,value) pair to republish it every 24 hours. Otherwise, (key,value) pairs expire 24 hours after publication, so as to limit stale index information in the system. For other applications, such as digital certificates or cryptographic hash to value mappings, longer expiration times may be appropriate.*"

Republishing originator data is handled in a timer event that resends the key-values in the originator's storage.

Code Listing 79: OriginatorRepublishElapsed

```
protected void OriginatorRepublishElapsed(object sender, ElapsedEventArgs
e)
{
  DateTime now = DateTime.Now;


  originatorStorage.Keys.Where(
    k => (now - originatorStorage.GetTimeStamp(k)).TotalMilliseconds >=
      Constants.ORIGINATOR_REPUBLISH_INTERVAL).ForEach(k =>
  {
```

```
    ID key = new ID(k);

    // Just use close contacts, don't do a lookup.

    var contacts = node.BucketList.GetCloseContacts(key,
node.OurContact.ID);


    contacts.ForEach(c =>

    {

      RpcError error = c.Protocol.Store(ourContact, key,
originatorStorage.Get(key));

      HandleError(error, c);

    });


    originatorStorage.Touch(k);

  });

}
```

Republished key-values persist for 24 hours.

Storing key-values onto the new node when a new node registers

From the spec: "*When a new node joins the system, it must store any key-value pair to which it is one of the k closest. Existing nodes, by similarly exploiting complete knowledge of their surrounding subtrees, will know which key-value pairs the new node should store. Any node learning of a new node therefore issues STORE RPCs to transfer relevant key-value pairs to the new node. To avoid redundant STORE RPCs, however, a node only transfers a key-value pair if it's own ID is closer to the key than are the IDs of other nodes.*"

Interpretation:

A new node (contact) will be instructed to store key-values that exist on the bootstrapping node (the one it's boostrapping with) for key-values that meet the following condition: The key XOR'd with the bootstrapping node's ID < (closer than) the key XOR'd the IDs of other nodes.

What does “other nodes” mean? Are these all other contacts the bootstrapping node knows about, or just the k closest contacts in the joining node’s bucket, or some other interpretation? We have to understand what “exploiting complete knowledge of their surrounding subtrees” means. First, this indicates that it isn’t just the joining node’s bucket. It would make sense to interpret this as “store the values onto the joining node for any key-value where the joining node will be closer to that key *when there are no other nodes that are closer.*” If the joining node becomes the *closest* node to a key-value, then it is requested to store that key-value.

It’s interesting to note that this algorithm executes regardless of whether the bootstrapping node actually added the the joining node to a k-bucket. Remember also that “joining” actually means contacting another node with any one of the four RPC calls. When a new node registers, republished key-values persist for 24 hours.

The implementation is as follows.

Code Listing 80: SendKeyValuesIfNewContact

```
/// <summary>
/// For a new contact, we store values to that contact whose keys ^
ourContact
/// are less than stored keys ^ [otherContacts].
/// </summary>
protected void SendKeyValuesIfNewContact(Contact sender)
{
  List<Contact> contacts = new List<Contact>();

  if (IsNewContact(sender))
  {
    lock (bucketList)
    {
      // Clone so we can release the lock.
      contacts = new List<Contact>(bucketList.Buckets.SelectMany(b =>
b.Contacts));
    }

    if (contacts.Count() > 0)
    {
```

```
      // and our distance to the key < any other contact's distance to
the key...
      storage.Keys.AsParallel().ForEach(k =>
      {
        // our min distance to the contact.
        var distance = contacts.Min(c => k ^ c.ID);

        // If our contact is closer, store the contact on its node.
        if ((k ^ ourContact.ID) < distance)
        {
          var error = sender.Protocol.Store(ourContact, new ID(k),
storage.Get(k));
          dht?.HandleError(error, sender);
        }
      });
    }
  }
}
```

Annoyingly, for every stored value, there just isn't any way to avoid performing the XOR computation on every contact. This could get expensive, and it is currently optimized using Linq's parallel feature.

Determining whether a contact is new is slightly more complicated than one would think. We need to check not only whether the contact exists in any of our buckets, but also whether it's a pending contact—one that wasn't placed in a bucket because the bucket was full, but nonetheless has already received any closer keys.

Code Listing 81: IsNewContact

```
/// <summary>
/// Returns true if the contact isn't in the bucket list or the
/// pending contacts list.
/// </summary>
protected bool IsNewContact(Contact sender)
```

```
{
  bool ret;

  lock (bucketList)
  {
    // If we have a new contact...
    ret = bucketList.ContactExists(sender);
  }

  if (dht != null)            // for unit testing, dht may be null
  {
    lock (dht.PendingContacts)
    {
      ret |= dht.PendingContacts.ContainsBy(sender, c => c.ID);
    }
  }

  return !ret;
}
```

Over-caching

From the spec: "*To avoid 'over-caching,' we make the expiration time of a (key,value) pair in any node's database exponentially inversely proportional to the number of nodes between the current node and the node whose ID is closest to the key ID.*"

- Inversely proportional*:* Meaning that the expiration time is shorter the more nodes that are between the current node and the closest node.
- Exponentially inversely proportional: Meaning the expiration time is a lot shorter with the more nodes that are between the current node and closest node.

The specification provides no guidance for what the calculation for “exponentially inversely proportional” should actually be. It’s also undefined as to what the time constants are—what is a baseline time for which a key-value should persist? It is assumed that this should be a maximum of 24 hours. We also need to track an expiration time that is separate from the key-value republish timestamp. Furthermore, up to this point, I haven’t implemented the concept of accelerated lookup optimization, which is where the value of *b* comes from. In this implementation, where we have bucket ranges, rather than a bucket-per-bit in the key space, the accelerated lookup optimization is irrelevant, so we’ll use **b==5** which is the spec’s recommended value for that optimization.

Also, who does the computation “between the current node and the node whose ID is closest to the key ID?” Is the current node:

- The sender that is caching the key-value on another code and counts the number of nodes between itself and receiving node?
- The receiver that is handling the store request and counts the number of nodes between itself and the sender node?

As discussed earlier, the entire concept of having separate stores (originator, republished, cached) is never discussed in the Kademlia specification. Without understanding these three different stores, trying to understand how caching works is probably impossible.

Caching occurs in only one place—when a value being looked up (and successfully found) is stored on a “close” node:

Code Listing 82: Handling Over-Caching

```
public (bool found, List<Contact> contacts, string val) FindValue(ID key)
{
  ...
  var lookup = router.Lookup(key, router.RpcFindValue);

  if (lookup.found)
  {
    ret = (true, null, lookup.val);
    // Find the first close contact (other than the one the value was
found by)
    // in which to *cache* the key-value.
    var storeTo = lookup.contacts.Where(c => c != lookup.foundBy).
      OrderBy(c => c.ID ^ key).FirstOrDefault();

    if (storeTo != null)
```

```
    {
      int separatingNodes = GetSeparatingNodesCount(ourContact, storeTo);
    int expTimeSec = (int)(Constants.EXPIRATION_TIME_SECONDS /
      Math.Pow(2, separatingNodes));
    RpcError error = storeTo.Protocol.Store(node.OurContact, key,
lookup.val,
      true, expTimeSec);
    HandleError(error, storeTo);
  }
}
```

Note the **`true`** flag, indicating that this RPC **`Store`** call is for caching purposes.

Never expiring republished key-values

It is reasonable for a cached key-value to expire, but we may never want to expire originator or republished key-values. One good example is a distributed blockchain (or distributed ledger) where data should never disappear, even if the original publisher disappears from the peer network. There are a variety of ways to do this, such as overriding, as in Code Listing 83.

Code Listing 83: Never Expiring Republished Key-Value

```
protected override void ExpireKeysElapsed(object sender, ElapsedEventArgs
e)
{
  RemoveExpiredData(cacheStorage);
  // RemoveExpiredData(republishStorage);
}
```

In a subclass of the **`Dht`** overriding this method, only the cached store expires.

Storing key-values onto the new node when a new node registers

There's a lot of setup here for creating two existing contacts and two key-values whose IDs have been specifically set. See the comments for the XOR distance "math."

Code Listing 84: TestNewContactGetsStoredContactsTest

```
[TestMethod]
public void TestNewContactGetsStoredContactsTest()
{
  // Set up a node at the midpoint.
  // The existing node has the ID 10000....
  Node existing = new Node(new Contact(null, ID.Mid), new
VirtualStorage());
  string val1 = "Value 1";
  string valMid = "Value Mid";

  // The existing node stores two items, one with an ID "hash" of 1,
  //the other with ID.Max
  // Simple storage, rather than executing the code for Store.
  existing.SimpleStore(ID.One, val1);
  existing.SimpleStore(ID.Mid, valMid);

  Assert.IsTrue(existing.Storage.Keys.Count == 2,
    "Expected the existing node to have two key-values.");

  // Create a contact in the existing node's bucket list that is closer
  // to one of the values.
  // This contact has the prefix 010000....
  Contact otherContact = new Contact(null, ID.Zero.SetBit(158));
  Node other = new Node(otherContact, new VirtualStorage());
  existing.BucketList.Buckets[0].Contacts.Add(otherContact);

  // The unseen contact has a prefix 0110000....
  VirtualProtocol unseenvp = new VirtualProtocol();
  Contact unseenContact = new Contact(unseenvp, ID.Zero.SetBit(157));
```

```
  Node unseen = new Node(unseenContact, new VirtualStorage());

  unseenvp.Node = unseen;     // final fixup.


  Assert.IsTrue(unseen.Storage.Keys.Count == 0,
    "The unseen node shouldn't have any key-values!");


  // An unseen node pings, and we should get back valMin only,
  // as ID.One ^ ID.Mid < ID.Max ^ ID.Mid

  existing.Ping(unseenContact);


  // Contacts             V1          V2

  // 10000000             00...0001   10...0000

  // 01000000


  // Math:

  // c1 ^ V1      c1 ^ V2      c2 ^ V1     c2 ^ V2

  // 100...001    000...000    010...001   110...000


  // c1 ^ V1 > c1 ^ V2, so v1 doesn't get sent to the unseen node.

  // c1 ^ V2 < c2 ^ V2, so it does get sent.


  Assert.IsTrue(unseen.Storage.Keys.Count == 1,
    "Expected 1 value stored in our new node.");

  Assert.IsTrue(unseen.Storage.Contains(ID.Mid),
    "Expected valMid to be stored.");

  Assert.IsTrue(unseen.Storage.Get(ID.Mid) == valMid,
    "Expected valMid value to match.");

}
```

Other optimizations

Ping is simply a “respond back with the random ID” that was sent. Internally, the buckets are potentially updated, and if the contact is new, store RPC calls are made to it for any values that it should store, as discussed above when a new node registers.

Piggy-backed ping

In his paper, Bruno Spori writes:[16]

“The situation is different when the first message a node received is a request message. In this case, the receiver cannot be sure whether the sender’s contact information [is] correct. It could be that the request was faked. To determine this, the piggy-backed ping is used. The effect of the piggy-backed ping is that the original sender of the request must send a ping reply upon receiving the reply message. Thus, the receiver of the request message is able to determine the correctness of the sender as well.”

We will instead rely on the error-handling mechanism for evicting contacts that do not respond or respond incorrectly or with errors. Error handling will be discussed later.

[16] http://pub.tik.ee.ethz.ch/students/2006-So/SA-2006-19.pdf

Chapter 11 Persisting the DHT

The bucket lists and contacts in each bucket need to be persisted so the last known state of the DHT can be restored. This is baked into the **Dht** implementation, serializing the data in a JSON file. The persistence of key-values is handled separately, and is defined by the specific implementation needs. Note that the **VirtualStorage** class provided in the baseline code does not persist its data. Internally, various properties are decorated with the **JsonIgnore** attribute to prevent circular serialization, and some classes have parameter-less public constructors for deserialization.

Serializing

This is straightforward—the only trick is enabling the type name handling in **Newtonsoft.Json** so that properties with abstract and interface types also serialize their concrete type.

Code Listing 85: Save the DHT

```
/// <summary>
/// Returns a JSON string of the serialized DHT.
/// </summary>
public string Save()
{
  var settings = new JsonSerializerSettings();
  settings.TypeNameHandling = TypeNameHandling.Auto;
  string json = JsonConvert.SerializeObject(this, Formatting.Indented,
settings);

  return json;
}
```

Deserializing

The deserializer is equally simple, however note the call to **DeserializationFixups**. This reduces the size of the JSON by not serializing certain properties that can be obtained from other properties. As a result, some minor fixups are necessary.

Code Listing 86: Load the DHT

```
public static Dht Load(string json)
{
  var settings = new JsonSerializerSettings();
  settings.TypeNameHandling = TypeNameHandling.Auto;

  Dht dht = JsonConvert.DeserializeObject<Dht>(json, settings);
  dht.DeserializationFixups();
  dht.SetupTimers();

  return dht;
}

protected void DeserializationFixups()
{
  ID = ourContact.ID;
  protocol = ourContact.Protocol;
  node = router.Node;
  node.OurContact = ourContact;
  node.BucketList.OurID = ID;
  node.BucketList.OurContact = ourContact;
  router.Dht = this;
  node.Dht = this;
}
```

DHT serialization unit test

The unit test is a simple test that a contact in a bucket gets persisted and restored correctly.

Code Listing 87: DhtSerializationTest

```
[TestMethod]
public void DhtSerializationTest()
{
  TcpSubnetProtocol p1 = new TcpSubnetProtocol(“http://127.0.0.1”, 2720,
1);
  TcpSubnetProtocol p2 = new TcpSubnetProtocol(“http://127.0.0.1”, 2720,
2);
  VirtualStorage store1 = new VirtualStorage();
  VirtualStorage store2 = new VirtualStorage();

  // Ensures that all nodes are closer, because ID.Max ^ n < ID.Max when
n > 0.
  Dht dht = new Dht(ID.Max, p1, new Router(), store1, store1, new
VirtualStorage());

  ID contactID = ID.Mid;      // a closer contact.
  Contact otherContact = new Contact(p2, contactID);
  Node otherNode = new Node(otherContact, store2);

  // Add this other contact to our peer list.
  dht.Node.BucketList.AddContact(otherContact);

  string json = dht.Save();

  Dht newDht = Dht.Load(json);
  Assert.IsTrue(newDht.Node.BucketList.Buckets.Sum(b => b.Contacts.Count)
== 1,
    "Expected our node to have 1 contact.");
  Assert.IsTrue(newDht.Node.BucketList.ContactExists(otherContact),
    "Expected our contact to have the other contact.");
```

```
  Assert.IsTrue(newDht.Router.Node == newDht.Node,
    "Router node not initialized.");

}
```

When you look at the JSON, you suddenly realize that shared objects, particularly contacts, are deserialized into separate instances. Because there are assumptions in the code regarding "same instance," and also as a way of ensuring that we're comparing contacts correctly (using their IDs), the **`Contact`** class implements **`IComparable`** and **`operator ==`** and **`operator !=`** overloads.

Chapter 12 Considerations for an Asynchronous Implementation

Thread Safety

These entry points to the node must be re-entrant:

- **`Ping`**
- **`Store`**
- **`FindNode`**
- **`FindValue`**

The first issue is with add contacts and the bucket manipulation that occurs. Collections should not be modified or otherwise manipulated while they are searched. We've seen the use of **`lock`** statements in previous code to ensure that collections are not modified asynchronously.

There are potentially more optimized approaches, such as locking only the specific **`KBucket`** being manipulated and only locking the **`BucketList`** when it itself is being modified; however, I will leave those for another time.

It is also assumed that the storage implementation can be re-entrant. In the virtual storage, this is handled by **`ConcurrentDictionary`** instances, for example:

```
protected ConcurrentDictionary<BigInteger, StoreValue> store;
```

Parallel queries

From the spec: "*The initiator then sends parallel, asynchronous find_node RPCs to the a [sic] nodes it has chosen, a is a system-wide concurrency parameter, such as 3.*"

In the lookup algorithm, Kademlia uses parallel, asynchronous queries to reduce timeout delays from failed nodes. Waiting for at least some of the nodes to respond in each batch of three closest nodes gives the system a chance to get even closer nodes to those first set of close nodes with the hope of acquiring k closer contacts without having to explore farther contacts.

It's not particularly clear why all the k closer contacts aren't queried in parallel to start with. Maybe the idea is that you want to try to get closer contacts from already close contacts. Certainly all the contacts could be queried and from the ones that respond first, we can select k closer ones. On the other hand, querying all the contacts simultaneously probably results in unnecessary network traffic as many of the FindNode RPC calls will be ignored.

The BaseRouter abstract class

For unit testing, it's useful to keep the nonparallel implementation, but ideally, both parallel and nonparallel calls to the **`Router`** should be made in the same way. An abstract **`BaseRouter`** class allows for this.

Code Listing 88: The BaseRouter Class

```
public abstract class BaseRouter

{
  public abstract
    (bool found, List<Contact> contacts, Contact foundBy, string val)
Lookup(
      ID key,

      Func<ID, Contact, (List<Contact> contacts, Contact foundBy, string
val)>
        rpcCall,

      bool giveMeAll = false);

}
```

The ParallelRouter

The **`ParallelRouter`** queues contacts to query, in addition to some other information each thread needs to know about when executing the RPC call.

Code Listing 89: ContactQueueItem

```
public class ContactQueueItem

{

  public ID Key { get; set; }

  public Contact Contact { get; set; }

  public Func<ID, Contact, (List<Contact> contacts, Contact foundBy,
string val)>
    RpcCall { get; set; }

  public List<Contact> CloserContacts { get; set; }

  public List<Contact> FartherContacts { get; set; }

  public FindResult FindResult { get; set; }

}
```

The **ParallelRouter** also initializes an internal thread pool.

Code Listing 90: InitializeThreadPool

```
protected void InitializeThreadPool()
{
  threads = new List<Thread>();
  Constants.MAX_THREADS.ForEach(() =>
  {
    Thread thread = new Thread(new ThreadStart(RpcCaller));
    thread.IsBackground = true;
    thread.Start();
  });
}
```

Work is queued and a semaphore is released for a thread to pick up the work.

Code Listing 91: QueueWork

```
protected void QueueWork(
  ID key,
  Contact contact,
  Func<ID, Contact, (List<Contact> contacts, Contact foundBy, string
val)> rpcCall,
  List<Contact> closerContacts,
  List<Contact> fartherContacts,
  FindResult findResult
)
{
  contactQueue.Enqueue(new ContactQueueItem()
  {
    Key = key,
    Contact = contact,
```

```
    RpcCall = rpcCall,
    CloserContacts = closerContacts,
    FartherContacts = fartherContacts,
    FindResult = findResult
 });

  Semaphore.Release();
}
```

Each thread dequeues an item for work in the **`RpcCaller`** method.

Code Listing 92: RpcCaller

```
protected void RpcCaller()
{
  while (true)
  {
    semaphore.WaitOne();
    ContactQueueItem item;

    if (contactQueue.TryDequeue(out item))
    {
      string val;
      Contact foundBy;

      if (GetCloserNodes(
        item.Key,
        item.Contact,
        item.RpcCall,
        item.CloserContacts,
        item.FartherContacts,
```

```
        out val,
        out foundBy))
      {
        if (!stopWork)
        {
          // Possible multiple "found"
          lock (locker)
          {
            item.FindResult.Found = true;
            item.FindResult.FoundBy = foundBy;
            item.FindResult.FoundValue = val;
            item.FindResult.FoundContacts = new
List<Contact>(item.CloserContacts);
          }
        }
      }
    }
  }
}
```

The salient point with the previous code is that when a value is found, it takes a snapshot of the current closer contacts and stores all the information about a closer contact in fields belonging to the **ParallelLookup** class.

The **ParallelRouter** must terminate its search after a certain amount of time, which handles unresponsive contacts. Whenever a response is received and new contacts are added to the list of contacts that can be queried, a timer is reset. The **Lookup** call exits when a value is found (for **FindValue**), or k closer contacts have been found, or the time period expires.

Code Listing 93: SetQueryTime

```
/// <summary>
/// Sets the time of the query to now.
/// </summary>
```

```
protected void SetQueryTime()
{
  now = DateTime.Now;
}

/// <summary>
/// Returns true if the query time has expired.
/// </summary>
protected bool QueryTimeExpired()
{
  return (DateTime.Now - now).TotalMilliseconds > Constants.QUERY_TIME;
}
```

The **Lookup** inner loop is the where the work is done, as with the nonparallel version, but notice instead how work is queued and we wait for responses—particularly the check for whether we've waited long enough in the **haveWork** assignment.

Code Listing 94: haveWork

```
...
ret.AddRangeDistinctBy(closerContacts, (a, b) => a.ID == b.ID);


// Spec: The lookup terminates when the initiator has queried and gotten
responses from the k closest nodes it has seen.
while (ret.Count < Constants.K && haveWork)
{
  Thread.Sleep(Constants.RESPONSE_WAIT_TIME);

  if (ParallelFound(findResult, ref foundReturn))
  {
    StopRemainingWork();

    return foundReturn;
```

```
  }


  List<Contact> closerUncontactedNodes =
    closerContacts.Except(contactedNodes).ToList();

  List<Contact> fartherUncontactedNodes =
    fartherContacts.Except(contactedNodes).ToList();

  bool haveCloser = closerUncontactedNodes.Count > 0;

  bool haveFarther = fartherUncontactedNodes.Count > 0;


  haveWork = haveCloser || haveFarther || !QueryTimeExpired();


  // Spec:  Of the k nodes the initiator has heard of closest to the
target...

  if (haveCloser)

  {

    // We're about to contact these nodes.

    var alphaNodes = closerUncontactedNodes.Take(Constants.ALPHA);

    contactedNodes.AddRangeDistinctBy(alphaNodes, (a, b) => a.ID ==
b.ID);

    alphaNodes.ForEach(n => QueueWork(
      key, n, rpcCall, closerContacts, fartherContacts, findResult));

    SetQueryTime();

  }

  else if (haveFarther)

  {

    // We're about to contact these nodes.

    var alphaNodes = fartherUncontactedNodes.Take(Constants.ALPHA);

    contactedNodes.AddRangeDistinctBy(alphaNodes, (a, b) => a.ID ==
b.ID);
```

```
    alphaNodes.ForEach(
      n => QueueWork(key, n, rpcCall, closerContacts, fartherContacts,
findResult));

    SetQueryTime();

  }

}
```

We can now take the **Dht** tests for the nonparallel version and create parallel versions of those tests, passing in the **ParallelRouter** instead:

```
Dht dht = new Dht(ID.RandomID, vp, () => new VirtualStorage(), new
ParallelRouter());
```

The result is that the parallel router unit tests also pass.

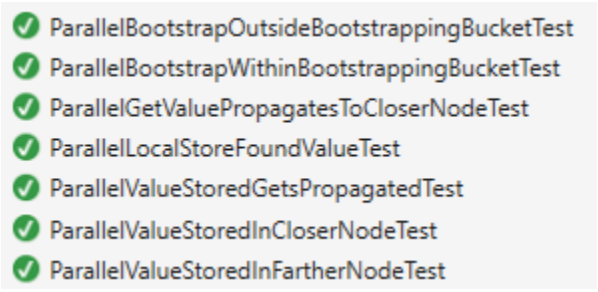

Figure 9

A potential problem occurs when there are threads still waiting for a response, and that response possibly occurs at some point after the **Lookup** method exits. We deal with this in several ways:

1. Remove all pending queued work with **DequeueRemainingWork();**
2. Copy the closer contacts collection when executing the **FindValue** RPC call: **foundContacts = item.CloserContacts.ToList();**
3. Set a “stop work” flag that ignores any further **FindValue** RPC successful return values for the current lookup.
4. Use a local (per-lookup call) **findResult** instance.
5. Use a common locker object to prevent both closer/farther contacts from being updated and update the values in the **findResult** instance.
6. Use local closer/farther/contacted lists per lookup, such that any remaining thread will update only the collections associated with its current work item, which is why **ContactQueueItem** includes the closer/farther collections as well as the **findResult** instance in which to store any found value.

This ensures that even if there are threads still performing work on a previous lookup, they do not affect the results of the current lookup.

While the use of a single **`locker`** object for blocking updates to collections and updating the find value is slightly inefficient, it avoids using nested locks; otherwise the thread, when if finds a value, would technically have to lock both the **`closerContacts`** collection and the **`findResult`** instance. Nested locks should be avoided. Also note that the **`Lookup`** method itself is not intended to be re-entrant.

Chapter 13 A Basic TCP Subnet Protocol

There's a few points to make in this implementation:

- A single port is used along with a subnet identifier to route requests to the correct handler. The subnet identifier makes it easier to test multiple "servers" on the same machine as we don't need to open (and "allow") a unique port number per "server."
- JSON is used as the serialization format for request and response data.
- RPC calls are synchronous, in that they wait for a response. There are other implementations that continue based on the random ID and handler, which I chose not to implement.

Request messages

Requests issued by the DHT are serialized to JSON from the following classes.

Code Listing 95: BaseRequest Class

```
public abstract class BaseRequest
{
  public object Protocol { get; set; }
  public string ProtocolName { get; set; }
  public BigInteger RandomID { get; set; }
  public BigInteger Sender { get; set; }

  public BaseRequest()
  {
     RandomID = ID.RandomID.Value;
  }
}

public class FindNodeRequest : BaseRequest
{
  public BigInteger Key { get; set; }
```

```
}

public class FindValueRequest : BaseRequest
{
  public BigInteger Key { get; set; }
}

public class PingRequest : BaseRequest { }

public class StoreRequest : BaseRequest
{
  public BigInteger Key { get; set; }
  public string Value { get; set; }
  public bool IsCached { get; set; }
  public int ExpirationTimeSec { get; set; }
}

public interface ITcpSubnet
{
  int Subnet { get; set; }
}

public class FindNodeSubnetRequest : FindNodeRequest, ITcpSubnet
{
  public int Subnet { get; set; }
}

public class FindValueSubnetRequest : FindValueRequest, ITcpSubnet
```

```
{
  public int Subnet { get; set; }
}

public class PingSubnetRequest : PingRequest, ITcpSubnet
{
  public int Subnet { get; set; }
}

public class StoreSubnetRequest : StoreRequest, ITcpSubnet
{
  public int Subnet { get; set; }
}
```

On the server side, which receives these messages, they are handled by a common request class.

Code Listing 96: CommonRequest—Server Side

```
/// <summary>
/// For passing to Node handlers with common parameters.
/// </summary>
public class CommonRequest
{
  public object Protocol { get; set; }
  public string ProtocolName { get; set; }
  public BigInteger RandomID { get; set; }
  public BigInteger Sender { get; set; }
  public BigInteger Key { get; set; }
  public string Value { get; set; }
  public bool IsCached { get; set; }
```

```
  public int ExpirationTimeSec { get; set; }
}
```

As the comment states, the common request simplifies the server implementation by having RPC handler methods with the same parameter.

Request handlers

The request handlers extract the pertinent pieces of the **`CommonRequest`** and call the appropriate method of the **`Node`** class. The important part here is that the contact protocol must be returned as part of the **`FindNode`** and **`FindValue`** response. Note that the returns are anonymous objects.

Code Listing 97: Request Handlers

```
public object ServerPing(CommonRequest request)
{
  IProtocol protocol = Protocol.InstantiateProtocol(
    request.Protocol, request.ProtocolName);
  Ping(new Contact(protocol, new ID(request.Sender)));


  return new { RandomID = request.RandomID };
}


public object ServerStore(CommonRequest request)
{
  IProtocol protocol = Protocol.InstantiateProtocol(
    request.Protocol, request.ProtocolName);
  Store(new Contact(protocol, new ID(request.Sender)),
    new ID(request.Key), request.Value, request.IsCached,
    request.ExpirationTimeSec);


  return new { RandomID = request.RandomID };
}
```

```
public object ServerFindNode(CommonRequest request)
{
  IProtocol protocol = Protocol.InstantiateProtocol(
    request.Protocol, request.ProtocolName);
  var (contacts, val) =
    FindNode(new Contact(protocol, new ID(request.Sender)), new
ID(request.Key));

  return new
  {
    Contacts = contacts.Select(c =>
    new
    {
      Contact = c.ID.Value,
      Protocol = c.Protocol,
      ProtocolName = c.Protocol.GetType().Name
    }).ToList(),
    RandomID = request.RandomID
  };
}

public object ServerFindValue(CommonRequest request)
{
  IProtocol protocol = Protocol.InstantiateProtocol(
    request.Protocol, request.ProtocolName);
  var (contacts, val) =
    FindValue(new Contact(protocol, new ID(request.Sender)), new
ID(request.Key));

  return new
```

```
  {
    Contacts = contacts?.Select(c =>
    new
    {
      Contact = c.ID.Value,
      Protocol = c.Protocol,
      ProtocolName = c.Protocol.GetType().Name
    })?.ToList(),
    RandomID = request.RandomID,
    Value = val
  };
}
```

Responses

JSON responses are deserialized into the following classes.

Code Listing 98: Server Responses

```
public abstract class BaseResponse
{
  public BigInteger RandomID { get; set; }
}

public class ErrorResponse : BaseResponse
{
  public string ErrorMessage { get; set; }
}

public class ContactResponse
```

```
{
  public BigInteger Contact { get; set; }
  public object Protocol { get; set; }
  public string ProtocolName { get; set; }
}

public class FindNodeResponse : BaseResponse
{
  public List<ContactResponse> Contacts { get; set; }
}

public class FindValueResponse : BaseResponse
{
  public List<ContactResponse> Contacts { get; set; }
  public string Value { get; set; }
}

public class PingResponse : BaseResponse { }

public class StoreResponse : BaseResponse { }
```

Server implementation

The server is a straightforward **`HttpListener`** implemented as a C# **`HttpListenerContext`** object, but note how the subnet ID is used to route the request to the specific node associated with the subnet.

Code Listing 99: ProcessRequest

```
protected override async void ProcessRequest(HttpListenerContext context)
{
```

```
  string data = new StreamReader(context.Request.InputStream,
    context.Request.ContentEncoding).ReadToEnd();


  if (context.Request.HttpMethod == "POST")

  {

    Type requestType;

    string path = context.Request.RawUrl;

    // Remove "//"

    // Prefix our call with "Server" so that the method name is
unambiguous.

    string methodName = "Server" + path.Substring(2);


    if (routePackets.TryGetValue(path, out requestType))

    {

      CommonRequest commonRequest =
        JsonConvert.DeserializeObject<CommonRequest>(data);
      int subnet = ((ITcpSubnet)JsonConvert.DeserializeObject(
        data, requestType)).Subnet;

      INode node;


      if (subnets.TryGetValue(subnet, out node))

      {

        await Task.Run(() =>
          CommonRequestHandler(methodName, commonRequest, node,
context));

      }

      else

      {

        SendErrorResponse(context, new ErrorResponse()
          { ErrorMessage = "Method not recognized." });

      }
```

```
    }
    else
    {
      SendErrorResponse(context, new ErrorResponse()
        { ErrorMessage = "Subnet node not found." });
    }

    context.Response.Close();
  }
}
```

The TCP subnet protocol handlers

The protocol implements the four RPC calls, issuing **`HTTP POST`**s to the server. Notice how the protocol is instantiated from the JSON return, and if the protocol isn't supported, the contact is removed from the contacts returned by **`FindNode`** and **`FindValue`**.

Code Listing 100: FindNode, FindValue, Ping, and Store Handlers

```
public (List<Contact> contacts, RpcError error) FindNode(Contact sender,
ID key)
{
  ErrorResponse error;
  ID id = ID.RandomID;
  bool timeoutError;

  var ret = RestCall.Post<FindNodeResponse, ErrorResponse>(
    url + ":" + port + "//FindNode",
  new FindNodeSubnetRequest()
  {
    Protocol = sender.Protocol,
    ProtocolName = sender.Protocol.GetType().Name,
```

```
    Subnet = subnet,
    Sender = sender.ID.Value,
    Key = key.Value,
    RandomID = id.Value
  }, out error, out timeoutError);

  try
  {
    var contacts = ret?.Contacts?.Select(
      val => new Contact(Protocol.InstantiateProtocol(
        val.Protocol, val.ProtocolName), new ID(val.Contact))).ToList();

    // Return only contacts with supported protocols.
    return (contacts?.Where(c => c.Protocol != null).ToList() ??
EmptyContactList(),
    GetRpcError(id, ret, timeoutError, error));
  }
  catch (Exception ex)
  {
    return (null, new RpcError() { ProtocolError = true,
      ProtocolErrorMessage = ex.Message });
  }
}

/// <summary>
/// Attempt to find the value in the peer network.
/// </summary>
/// <returns>A null contact list is acceptable here as it is a valid
return
/// if the value is found.
```

```
/// The caller is responsible for checking the timeoutError flag to make
/// sure null contacts is not

/// the result of a timeout error.</returns>

public (List<Contact> contacts, string val, RpcError error)
  FindValue(Contact sender, ID key)

{

  ErrorResponse error;

  ID id = ID.RandomID;

  bool timeoutError;


  var ret = RestCall.Post<FindValueResponse, ErrorResponse>(
    url + ":" + port + "//FindNode",

  new FindValueSubnetRequest()

  {

    Protocol = sender.Protocol,

    ProtocolName = sender.Protocol.GetType().Name,

    Subnet = subnet,

    Sender = sender.ID.Value,

    Key = key.Value,

    RandomID = id.Value

  }, out error, out timeoutError);


  try

  {

    var contacts = ret?.Contacts?.Select(
      val => new Contact(Protocol.InstantiateProtocol(val.Protocol,
        val.ProtocolName), new ID(val.Contact))).ToList();


    // Return only contacts with supported protocols.

    return (contacts?.Where(c => c.Protocol != null).ToList(),
      ret.Value, GetRpcError(id, ret, timeoutError, error));
```

```
  }
  catch (Exception ex)
  {
    return (null, null, new RpcError() { ProtocolError = true,
      ProtocolErrorMessage = ex.Message });
  }
}

public RpcError Ping(Contact sender)
{
  ErrorResponse error;
  ID id = ID.RandomID;
  bool timeoutError;

  var ret = RestCall.Post<FindValueResponse, ErrorResponse>(
    url + ":" + port + "//Ping",
  new PingSubnetRequest()
  {
    Protocol = sender.Protocol,
    ProtocolName = sender.Protocol.GetType().Name,
    Subnet = subnet,
    Sender = sender.ID.Value,
    RandomID = id.Value
  },
  out error, out timeoutError);

  return GetRpcError(id, ret, timeoutError, error);
}
```

```
public RpcError Store(Contact sender, ID key, string val, bool isCached =
false,
  int expirationTimeSec = 0)
{
  ErrorResponse error;
  ID id = ID.RandomID;
  bool timeoutError;

  var ret = RestCall.Post<FindValueResponse, ErrorResponse>(
    url + ":" + port + "//Store",
  new StoreSubnetRequest()
  {
    Protocol = sender.Protocol,
    ProtocolName = sender.Protocol.GetType().Name,
    Subnet = subnet,
    Sender = sender.ID.Value,
    Key = key.Value,
    Value = val,
    IsCached = isCached,
    ExpirationTimeSec = expirationTimeSec,
    RandomID = id.Value
  },
  out error, out timeoutError);

  return GetRpcError(id, ret, timeoutError, error);
}
```

The **RpcError** class manages the kinds of errors that we can encounter, and is instantiated in the **GetRpcError** method.

Code Listing 101: Handling RPC Errors

```
public class RpcError

{

  public bool HasError
  {
    get { return TimeoutError || IDMismatchError || PeerError ||
ProtocolError; }
  }


  public bool TimeoutError { get; set; }

  public bool IDMismatchError { get; set; }

  public bool PeerError { get; set; }

  public bool ProtocolError { get; set; }

  public string PeerErrorMessage { get; set; }

  public string ProtocolErrorMessage { get; set; }

}

protected RpcError GetRpcError(
  ID id,
  BaseResponse resp,
  bool timeoutError,
  ErrorResponse peerError)

{

  return new RpcError()
  {
    IDMismatchError = id != resp.RandomID,
    TimeoutError = timeoutError,
    PeerError = peerError != null,
    PeerErrorMessage = peerError?.ErrorMessage
  };

}
```

Note that this class reflects several different errors that can occur:

- Timeout: The peer failed to respond in the **`RestCall.REQUEST_TIMEOUT`** period, which by default is 500 ms.

- ID mismatch: The peer responded, but not with an ID that matched the sender's random ID.
- Peer: The peer encountered an exception, in which case the exception message is returned to the caller.
- Deserialization: The **`Post`** method catches JSON deserialization errors, which also indicates an error with the peer response.

TCP subnet unit testing

As with the unit tests for the protocol itself, studying these unit tests is useful for how one sets up a server and client. The following unit tests validate the round-trip calls, exercising the protocol calls and the server. Each test initializes the server and then tears it down.

Code Listing 102: TCP Subnet Setup and Teardown

```
[TestClass]
public class TcpSubnetTests
{
  protected string localIP = "http://127.0.0.1";
  protected int port = 2720;
  protected TcpSubnetServer server;

  [TestInitialize]
  public void Initialize()
  {
    server = new TcpSubnetServer(localIP, port);
  }

  [TestCleanup]
  public void TestCleanup()
  {
    server.Stop();
  }
...
```

The unit tests exercise each of the four RPC calls as well as a timeout error.

PingRouteTest

This test verifies the **`Ping`** RPC call.

Code Listing 103: PingRouteTest

```
[TestMethod]
public void PingRouteTest()
{
  TcpSubnetProtocol p1 = new TcpSubnetProtocol(localIP, port, 1);
  TcpSubnetProtocol p2 = new TcpSubnetProtocol(localIP, port, 2);
  ID ourID = ID.RandomID;
  Contact c1 = new Contact(p1, ourID);
  Node n1 = new Node(c1, new VirtualStorage());
  Node n2 = new Node(new Contact(p2, ID.RandomID), new VirtualStorage());
  server.RegisterProtocol(p1.Subnet, n1);
  server.RegisterProtocol(p2.Subnet, n2);
  server.Start();

  p2.Ping(c1);
}
```

Oddly there's no assertion here, as nothing of note happens. The point of this is that no exceptions are thrown.

StoreRouteTest

This test verifies the **`Store`** RPC call.

Code Listing 104: StoreRoutTest

```
[TestMethod]
public void StoreRouteTest()
```

```
{
  TcpSubnetProtocol p1 = new TcpSubnetProtocol(localIP, port, 1);
  TcpSubnetProtocol p2 = new TcpSubnetProtocol(localIP, port, 2);
  ID ourID = ID.RandomID;
  Contact c1 = new Contact(p1, ourID);
  Node n1 = new Node(c1, new VirtualStorage());
  Node n2 = new Node(new Contact(p2, ID.RandomID), new VirtualStorage());
  server.RegisterProtocol(p1.Subnet, n1);
  server.RegisterProtocol(p2.Subnet, n2);
  server.Start();

  Contact sender = new Contact(p1, ID.RandomID);
  ID testID = ID.RandomID;
  string testValue = "Test";
  p2.Store(sender, testID, testValue);
  Assert.IsTrue(n2.Storage.Contains(testID),
    "Expected remote peer to have value.");
  Assert.IsTrue(n2.Storage.Get(testID) == testValue,
    "Expected remote peer to contain stored value.");
}
```

FindNodesRouteTest

This test verifies the **`FindNodes`** RPC call.

Code Listing 105: FindNodesRouteTest

```
[TestMethod]
public void FindNodesRouteTest()
{
  TcpSubnetProtocol p1 = new TcpSubnetProtocol(localIP, port, 1);
  TcpSubnetProtocol p2 = new TcpSubnetProtocol(localIP, port, 2);
```

```
  ID ourID = ID.RandomID;

  Contact c1 = new Contact(p1, ourID);

  Node n1 = new Node(c1, new VirtualStorage());

  Node n2 = new Node(new Contact(p2, ID.RandomID), new VirtualStorage());


  // Node 2 knows about another contact, that isn't us (because we're
excluded.)

  ID otherPeer = ID.RandomID;

  n2.BucketList.Buckets[0].Contacts.Add(new Contact(
    new TcpSubnetProtocol(localIP, port, 3), otherPeer));


  server.RegisterProtocol(p1.Subnet, n1);

  server.RegisterProtocol(p2.Subnet, n2);

  server.Start();


  ID id = ID.RandomID;

  List<Contact> ret = p2.FindNode(c1, id).contacts;


  Assert.IsTrue(ret.Count == 1, "Expected 1 contact.");

  Assert.IsTrue(ret[0].ID == otherPeer,
    "Expected contact to the other peer (not us).");

}
```

FindValueRouteTest

This test verifies the **`FindValue`** RPC call.

Code Listing 106: FindValueRouteTest

```
[TestMethod]

public void FindValueRouteTest()

{
```

```
  TcpSubnetProtocol p1 = new TcpSubnetProtocol(localIP, port, 1);
  TcpSubnetProtocol p2 = new TcpSubnetProtocol(localIP, port, 2);
  ID ourID = ID.RandomID;
  Contact c1 = new Contact(p1, ourID);
  Node n1 = new Node(c1, new VirtualStorage());
  Node n2 = new Node(new Contact(p2, ID.RandomID), new VirtualStorage());

  server.RegisterProtocol(p1.Subnet, n1);
  server.RegisterProtocol(p2.Subnet, n2);
  server.Start();

  ID testID = ID.RandomID;
  string testValue = "Test";
  p2.Store(c1, testID, testValue);

  Assert.IsTrue(n2.Storage.Contains(testID),
    "Expected remote peer to have value.");
  Assert.IsTrue(n2.Storage.Get(testID) == testValue,
    "Expected remote peer to contain stored value.");

  var ret = p2.FindValue(c1, testID);

  Assert.IsTrue(ret.contacts == null, "Expected to find value.");
  Assert.IsTrue(ret.val == testValue,
    "Value does not match expected value from peer.");
}
```

UnresponsiveNodeTest

This test verifies that an unresponsive node results in a timeout error.

Code Listing 107: UnresponsiveNodeTest

```
[TestMethod]
public void UnresponsiveNodeTest()
{
  TcpSubnetProtocol p1 = new TcpSubnetProtocol(localIP, port, 1);
  TcpSubnetProtocol p2 = new TcpSubnetProtocol(localIP, port, 2);
  p2.Responds = false;
  ID ourID = ID.RandomID;
  Contact c1 = new Contact(p1, ourID);
  Node n1 = new Node(c1, new VirtualStorage());
  Node n2 = new Node(new Contact(p2, ID.RandomID), new VirtualStorage());

  server.RegisterProtocol(p1.Subnet, n1);
  server.RegisterProtocol(p2.Subnet, n2);
  server.Start();

  ID testID = ID.RandomID;
  string testValue = "Test";
  RpcError error = p2.Store(c1, testID, testValue);

  Assert.IsTrue(error.TimeoutError, "Expected timeout.");
}
```

Chapter 14 RPC Error Handling and Delayed Eviction

One of the optimizations in the Kademlia protocol can now be implemented—delayed eviction. From the spec: "*When a Kademlia node receives an RPC from an unknown contact and the k-bucket for that contact is already full with k entries, the node places the new contact in a replacement cache of nodes eligible to replace stale k-bucket entries. The next time the node queries contacts in the k-bucket, any unresponsive ones can be evicted and replaced with entries in the replacement cache. The replacement cache is kept sorted by time last seen, with the most recently seen entry having the highest priority as a replacement candidate.*"

What happens when the peer throws an exception or the random ID that the peer responds with doesn't match what was sent? To make matters simple, in this implementation we'll handle all error conditions, including timeouts, in the same way.

The implementation for handling evictions and replacing them with contacts waiting to be added to the bucket is handled in the **`Dht`**. First, the error handler.

Code Listing 108: HandleError Method

```
/// <summary>
/// Put the timed out contact into a collection and increment the number
of
// times it has timed out.
/// If it has timed out a certain amount, remove it from the bucket and
/// replace it with the most
/// recent pending contact that are queued for that bucket.
/// </summary>
public void HandleError(RpcError error, Contact contact)
{
  // For all errors:
  int count = AddContactToEvict(contact.ID.Value);

  if (count == Constants.EVICTION_LIMIT)
  {
    ReplaceContact(contact);
```

```
  }
}
```

And the implementation of **`DelayEviction`**.

Code Listing 109: DelayEviction

```
/// <summary>
///  The contact that did not respond (or had an error) gets n tries
before
/// being evicted and replaced with the most recently contact that wants
to
/// go into the non-responding contact's kbucket.
/// </summary>
/// <param name="toEvict">The contact that didn't respond.</param>
/// <param name="toReplace">The contact that can replace
/// the non-responding contact.</param>
public void DelayEviction(Contact toEvict, Contact toReplace)
{
  // Non-concurrent list needs locking.
  lock (pendingContacts)
  {
    // Add only if it's a new pending contact.
    pendingContacts.AddDistinctBy(toReplace, c=>c.ID);
  }

  BigInteger key = toEvict.ID.Value;
  int count = AddContactToEvict(key);

  if (count == Constants.EVICTION_LIMIT)
  {
    ReplaceContact(toEvict);
```

```
  }
}
```

Note the difference:

- When a contact fails to respond with an RPC call, its eviction count is incremented, and if exceeded, it is removed and replaced with the most recently seen contact that goes in the bucket.
- Alternatively, when a contact is being added to a full bucket and the last seen contact fails to respond (or has an error), it is added to the eviction pool, and the new contact wanting to be added is placed into the pending contacts pool.

The rest is just the implementation of the helper methods.

Code Listing 110: Eviction Helper Methods

```
protected int AddContactToEvict(BigInteger key)
{
  if (!evictionCount.ContainsKey(key))
  {
    evictionCount[key] = 0;
  }

  int count = evictionCount[key] + 1;
  evictionCount[key] = count;

  return count;
}

protected void ReplaceContact(Contact toEvict)
{
  KBucket bucket = node.BucketList.GetKBucket(toEvict.ID);

  // Prevent other threads from manipulating the bucket list or buckets.
  lock (node.BucketList)
```

```
  {
    EvictContact(bucket, toEvict);
    ReplaceWithPendingContact(bucket);
  }
}

protected void EvictContact(KBucket bucket, Contact toEvict)
{
  evictionCount.TryRemove(toEvict.ID.Value, out _);

Validate.IsTrue<BucketDoesNotContainContactToEvict>(bucket.Contains(toEvict
.ID),
    "Bucket doesn't contain the contact to be evicted.");
  bucket.EvictContact(toEvict);
}

/// <summary>
/// Find a pending contact that goes into the bucket that now has room.
/// </summary>
protected void ReplaceWithPendingContact(KBucket bucket)
{
  Contact contact;

  // Nonconcurrent list needs locking while we query it.
  lock (pendingContacts)
  {
    contact = pendingContacts.Where(c => node.BucketList.GetKBucket(c.ID)
==
      bucket).OrderBy(c => c.LastSeen).LastOrDefault();
```

```
    if (contact != null)
    {
      pendingContacts.Remove(contact);
      bucket.AddContact(contact);
    }
  }
}
```

Chapter 15 Putting It Together: A Demo

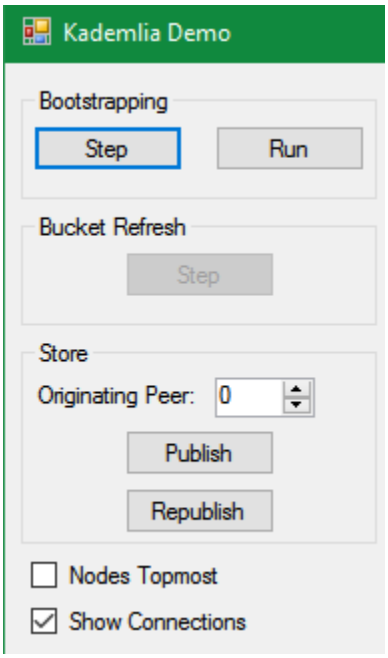

Figure 10

As usual, there's nothing like a visual demo to see what is happening. This demo uses my open-source diagramming tool FlowSharp[17] as the drawing canvas. As we see in Figure 11, we start with 60 peers (green), five of which are known peers (red).

Code Listing 111: Initializing DHTs and Known Peers

```
protected void InitializeDhts()
{
  dhts = new List<Dht>();
  dhtPos = new List<Rectangle>();
  peerColor = new Dictionary<BigInteger, Color>();

  NUM_DHT.ForEach((n) =>
  {
```

[17] https://github.com/cliftonm/FlowSharp

```
    IProtocol protocol = new TcpSubnetProtocol("http://127.0.0.1", 2720,
n);

    Dht dht = new Dht(ID.RandomID, protocol,
        () => new VirtualStorage(), new Router());

    peerColor[dht.ID.Value] = Color.Green;

    server.RegisterProtocol(n, dht.Node);

    dhts.Add(dht);

    dhtPos.Add(new Rectangle(XOFFSET + rnd.Next(-JITTER, JITTER) + (n %
ITEMS_PER_ROW) * XSPACING, YOFFSET + rnd.Next(-JITTER, JITTER) + (n /
ITEMS_PER_ROW) * YSPACING, SIZE, SIZE));

  });

}


protected void InitializeKnownPeers()

{

  knownPeers = new List<Dht>();

  List<Dht> workingList = new List<Dht>(dhts);


  NUM_KNOWN_PEERS.ForEach(() =>

  {

    Dht knownPeer = workingList[rnd.Next(workingList.Count)];

    peerColor[knownPeer.ID.Value] = Color.Red;

    knownPeers.Add(knownPeer);

    workingList.Remove(knownPeer);

  });

}
```

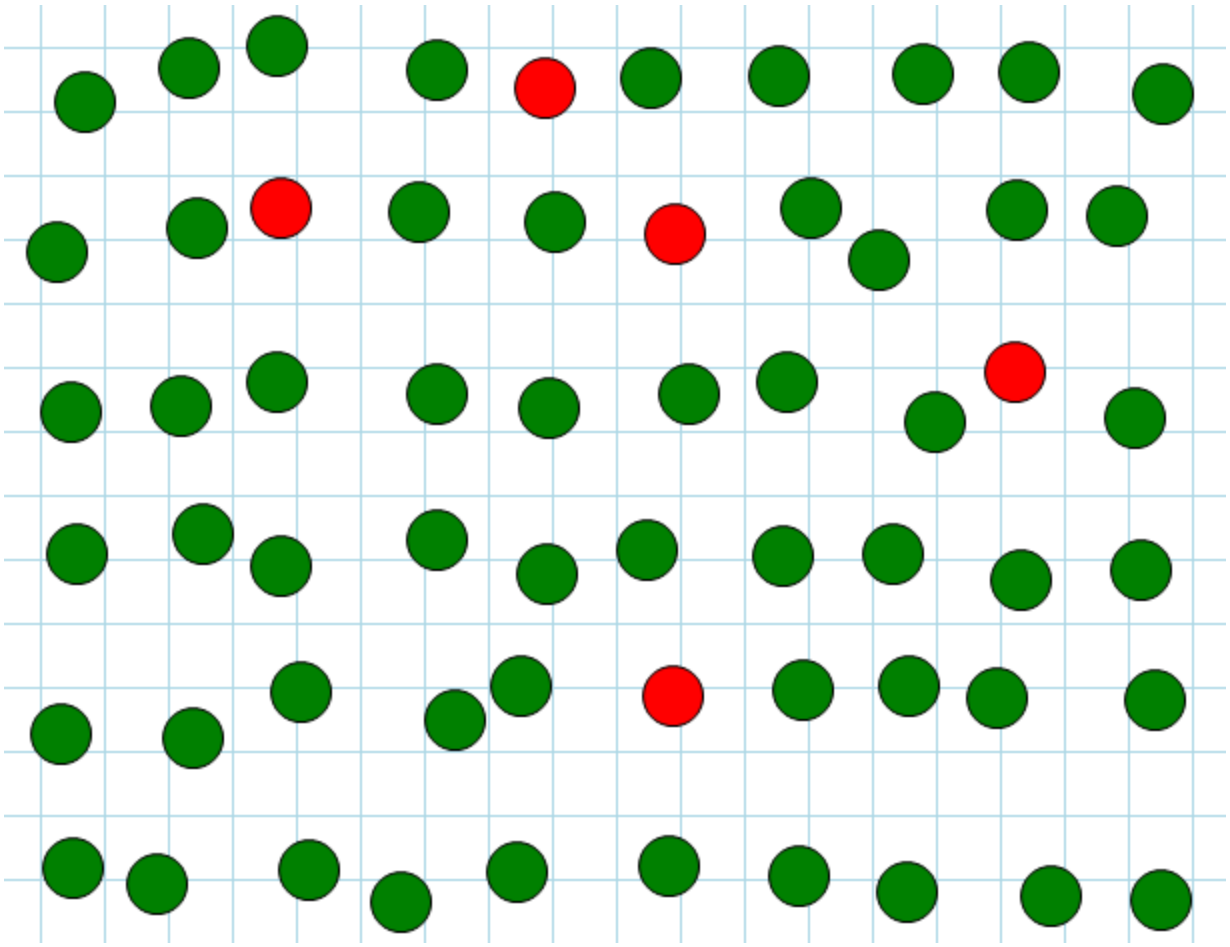

Figure 11

Bootstrapping

We can now bootstrap to a random peer.

Code Listing 112: Bootstrapping a Peer

```
protected void BootstrapWithAPeer(int peerBootstrappingIdx)
{
  Dht dht = dhts[peerBootstrappingIdx];
  var peerList = knownPeers.ExceptBy(dht, c => c.ID).ToList();
  Dht bootstrapWith = peerList[rnd.Next(peerList.Count)];
  dht.Bootstrap(bootstrapWith.Contact);
}
```

As Figure 12 shows, after each peer bootstraps with one of the known peers (randomly selected), the peer network is established.

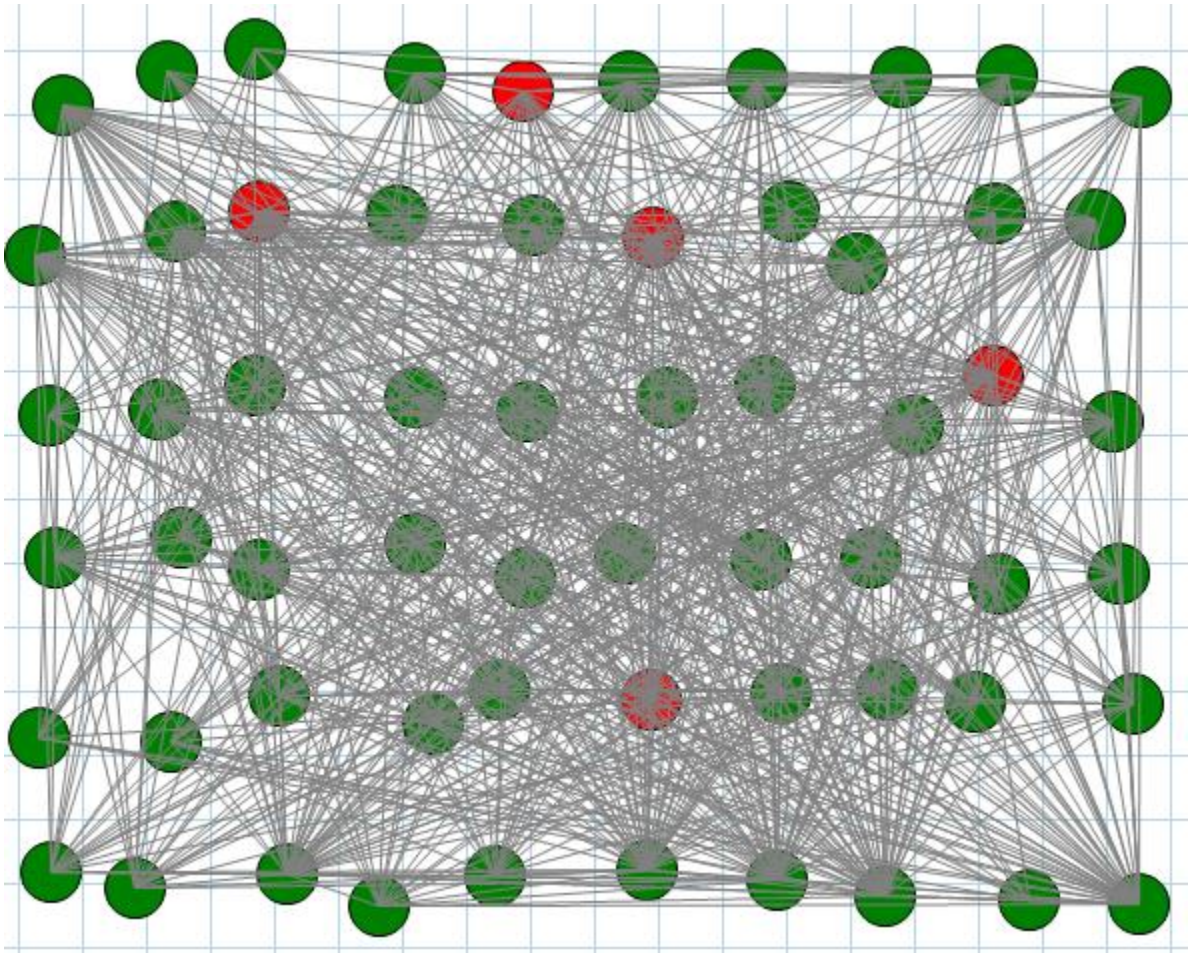

Figure 12

The directionality of the connection is not shown—arrows would get lost in this drawing!

Bucket Refresh

To illustrate bucket refresh, let's start with a smaller set of peers (25).

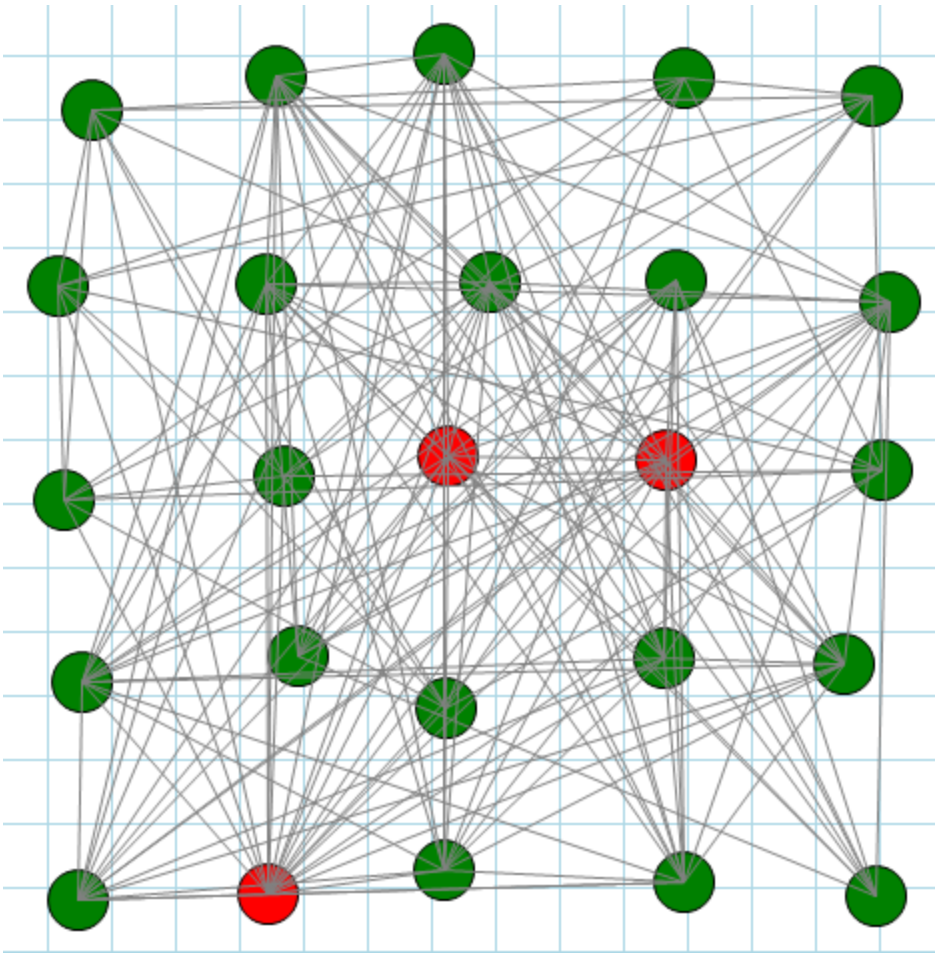

Figure 13

A bucket refresh calls **`FindNode`** all the contacts in each bucket. This updates the contacts for each peer based on the k closest contacts returned by the contact.

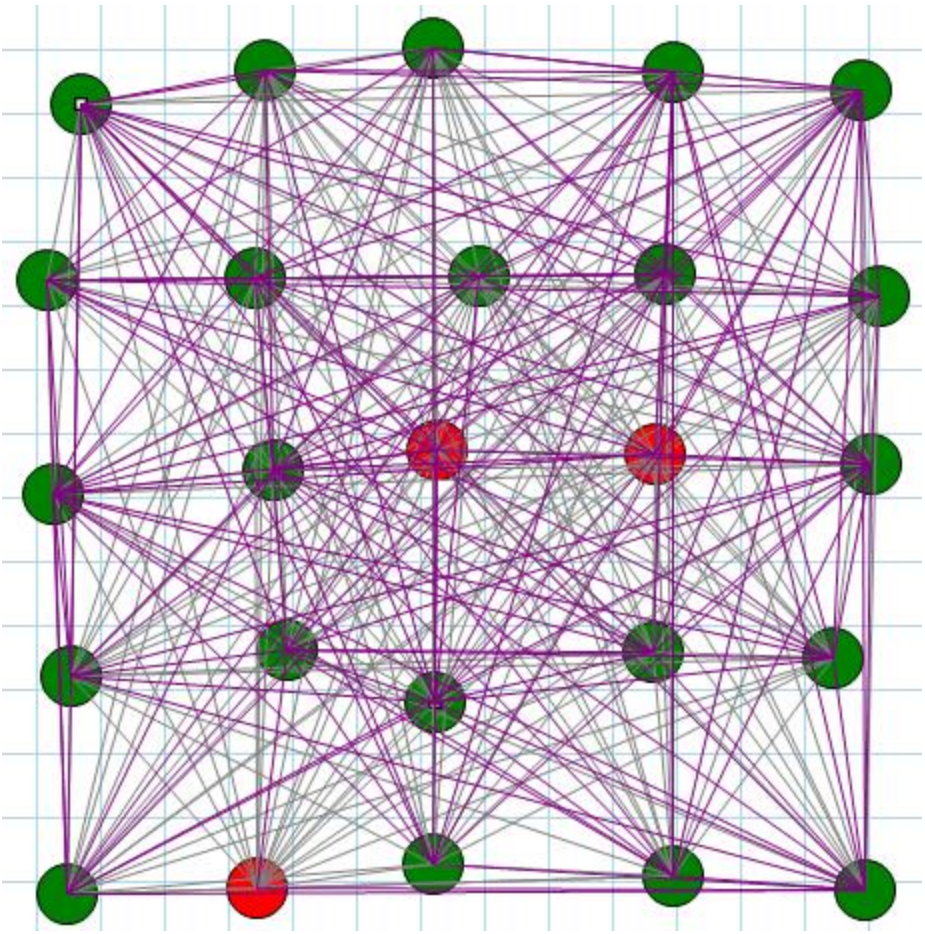

Figure 14

The newly discovered contacts are drawn in purple. In a small network like this, just about every peer learns about every other peer—another iteration of bucket refreshing results in only a couple more contacts being discovered.

Store value

When a noncached value is stored to a peer, it is republished to close peers. We can see this by coloring the originator with yellow, the immediate peer we're storing the value to in blue, and the peers to which the value is republished in orange.

Code Listing 113: Publish

```
/// <summary>
/// Color the originator with yellow
/// the immediate peer we're storing the value to in blue
```

```
/// and the peers to which the value is republished in orange:
/// </summary>
private void btnPublish_Click(object sender, EventArgs e)
{
  firstContacts = new List<Dht>();
  storeKey = ID.RandomID;
  originatorDht = dhts[(int)nudPeerNumber.Value];
  originatorDht.Store(storeKey, "Test");
  System.Threading.Thread.Sleep(500);
  dhts.Where(d => d.RepublishStorage.Contains(storeKey)).
    ForEach(d => firstContacts.Add(d));
  UpdatePeerColors();
  DrawDhts();
}
```

In a small network, because the store gets published to k peers, most of the peers are involved.

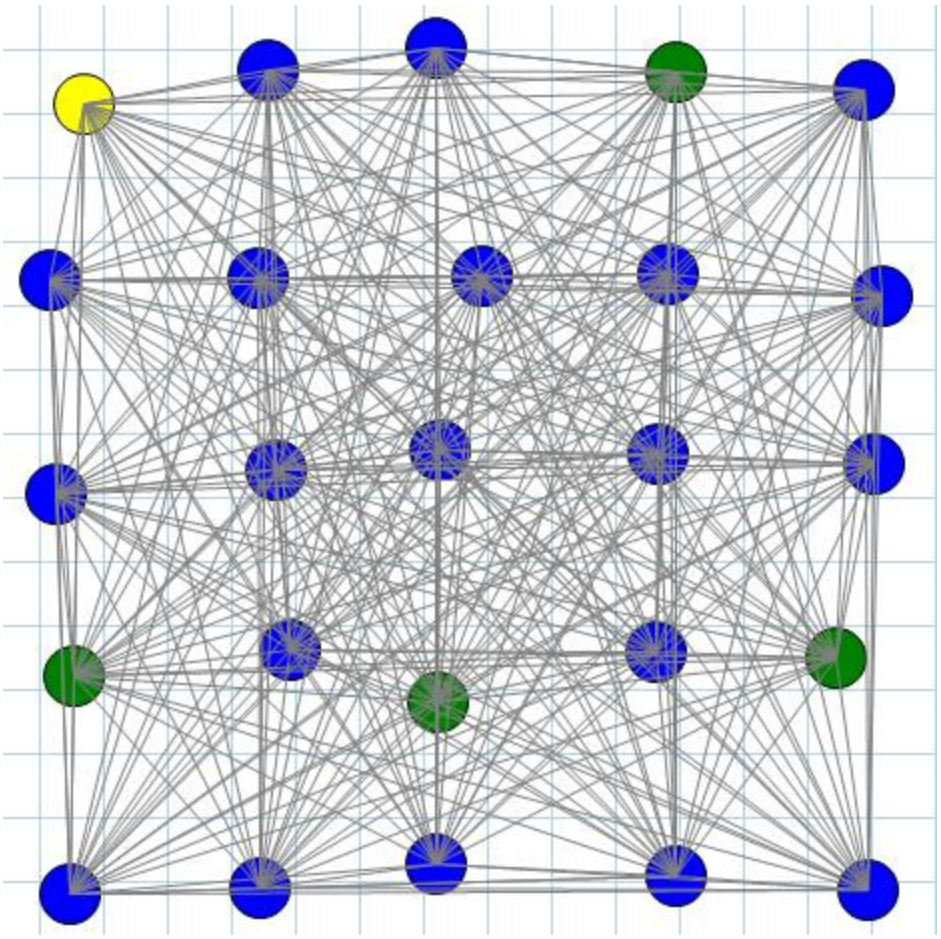

Figure 15

Store republish

When we force an immediate store republish, we see one node, which is a closer contact, getting the republished key-value.

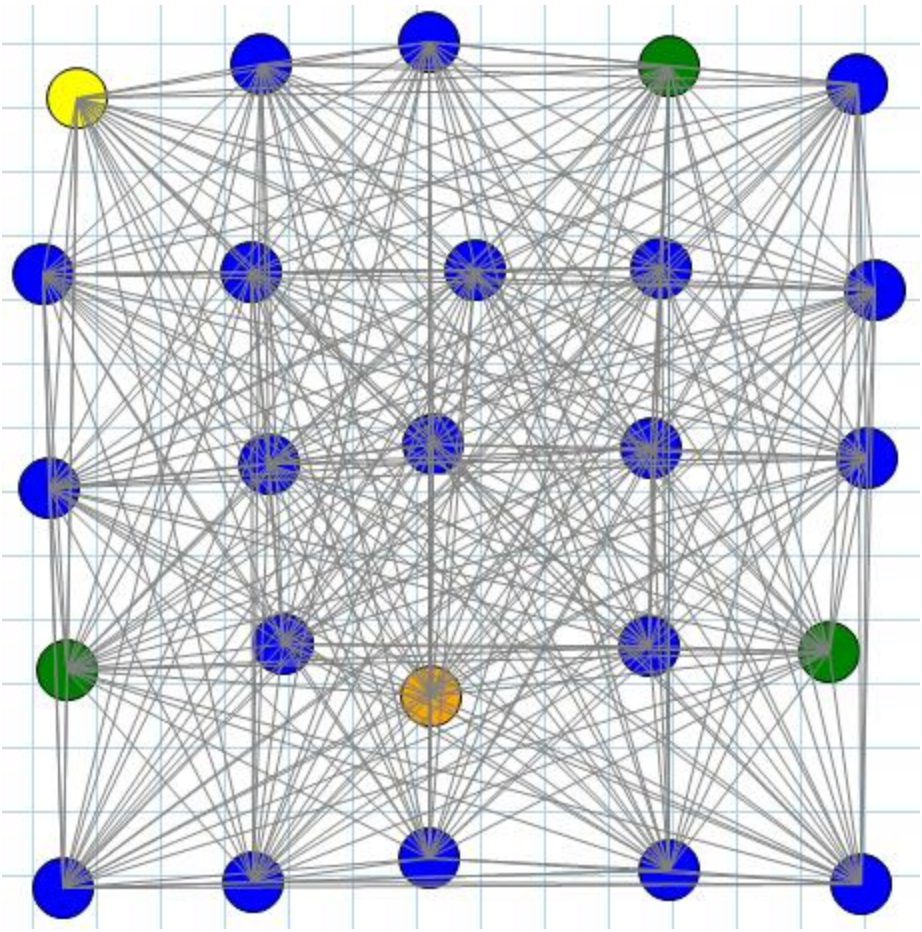

Figure 16

In a larger network, this becomes more obvious (nodes have been moved to the topmost in the rendering).

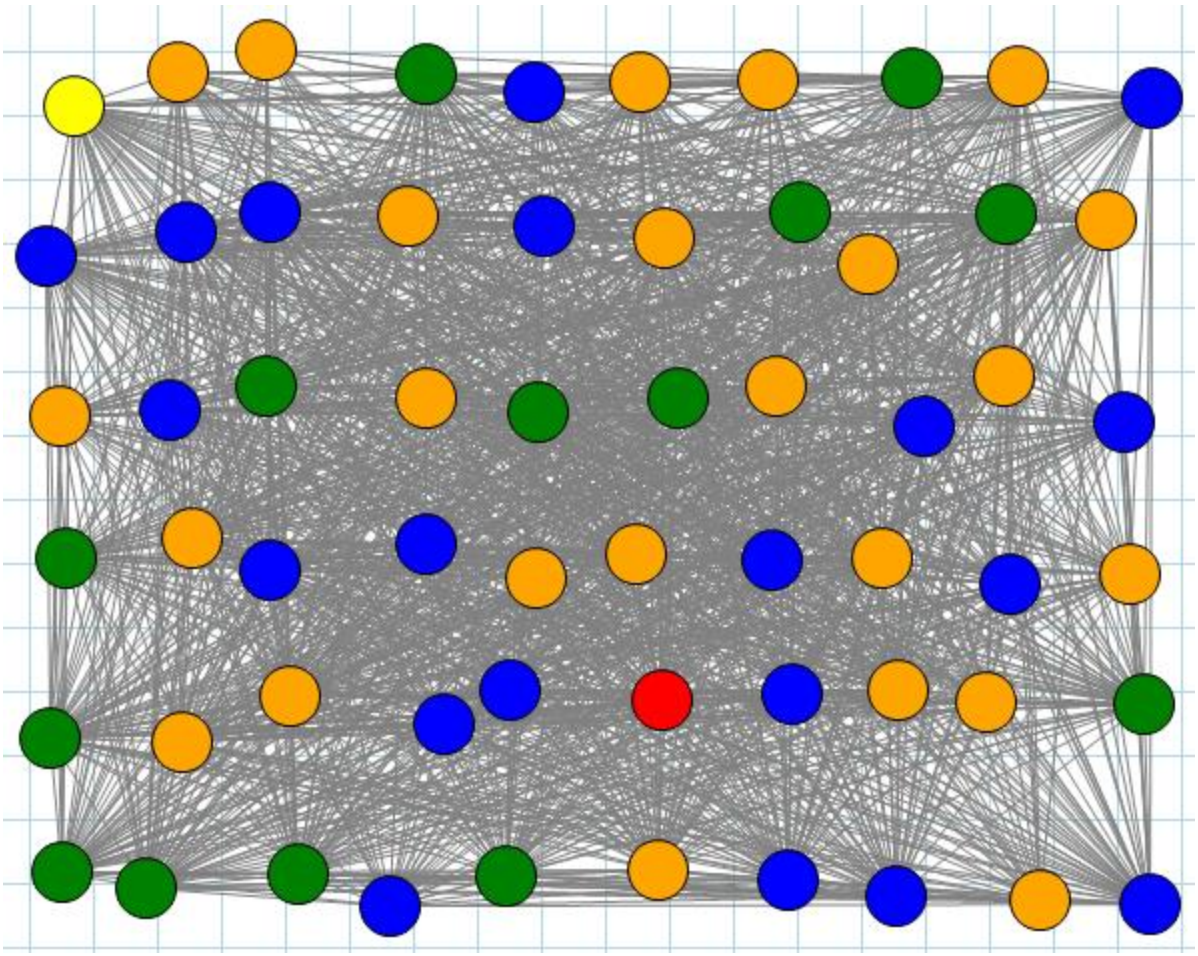

Figure 17

Remember that key-values are republished only on k closer contacts, so not every peer gets the republished key-value.

Chapter 16 Things Not Implemented

There are a few items from the specification not implemented here.

UDP dropouts

From the spec:

"*A related problem is that because Kademlia uses UDP, valid contacts will sometimes fail to respond when network packets are dropped. Because packet loss often indicates network congestion, Kademlia locks unresponsive contacts and avoids sending them any further RPCs for an exponentially increasing backoff interval. Because at most stages Kademlia's lookup only needs to hear from one of k nodes, the system typically does not retransmit dropped RPCs to the same node.*

When a contact fails to respond to 5 RPCs in a row, it is considered stale. If a fe-bucket is not full or its replacement cache is empty, Kademlia merely flags stale contacts rather than remove them. This ensures, among other things, that if a node's own network connection goes down temporarily, the node won't completely void all of its k-buckets."

This is true not just for UDP packets but any connection—it may go down for a while. This algorithm is somewhat entangled with delayed eviction. In delayed eviction, the spec state "any unresponsive ones can be evicted." It is the spec's description of UDP dropouts that actually defines what "unresponsive" actually means.

Accelerated lookups

Again, from the spec:

"*When a contact fails to respond to 5 RPCs in a row, it is considered stale. If a fe-bucket is not full or its replacement cache is empty, Kademlia merely flags stale contacts rather than remove them. This ensures, among other things, that if a node's own network connection goes down temporarily, the node won't completely void all of its k-buckets.*

Another optimization in the implementation is to achieve fewer hops per lookup by increasing the routing table size. Conceptually, this is done by considering IDs b bits at a time instead of just one bit at a time. As previously described, the expected number of hops per lookup is log2n. By increasing the routing table's size to an expected $2^b \log_{2^b} n$ *k-buckets, we can reduce the number of expected hops to* $\log_{2^b} n$.*"*

In this implementation we have bucket ranges rather than a bucket per prefix bits in the key space; therefore, the accelerated lookup optimization is irrelevant, because the bucket ranges typically span many prefix bits.

Sybil attacks

Peer-to-peer networks are vulnerable to Sybil attacks:

"In a Sybil attack, the attacker subverts the reputation system of a peer-to-peer network by creating a large number of pseudonymous identities, using them to gain a disproportionately large influence. A reputation system's vulnerability to a Sybil attack depends on how cheaply identities can be generated, the degree to which the reputation system accepts inputs from entities that do not have a chain of trust linking them to a trusted entity, and whether the reputation system treats all entities identically."[18]

In the current implementation, if the peer network is already well populated (most k-buckets are full) a Sybil attack would not replace "good," known peers—the attack would simply place the attempt into the DHT's pending contact buffer. In a mostly unpopulated network (most k-buckets have room for more peers), the subsequent failure to get a response from a peer would result in its eventual eviction. The piggyback ping approach is also a means for the recipient of the RPC call to verify the sender.

[18] https://en.wikipedia.org/wiki/Sybil_attack

Conclusion

Petar Maymounkov wrote: "...my opinion is that Kademlia is so simple, that with a modern language like Go, a good implementation of Kademlia (the algorithm) is no more than 100 lines of code."[19] Perhaps when looking at just the algorithm (the router in particular), this may be true, but there is a tremendous amount of detail that has to go into the architecture and implementation of the Kademlia protocol. As one dives deep into the spec, there are contradictions between the two versions, between implementations out there in the wild and the spec, and numerous ambiguities that must be resolved by carefully understanding the spec and carefully inspecting other implementations and descriptions of the protocol. Anyone adopting a library that implements the Kademlia protocol should have a thorough understanding of these contradictions and ambiguities, and also must go through any implementation with a fine-tooth comb to see how these are addressed—if they are addressed at all. In particular, when looking at other implementations, does the implementation (in no particular order of importance):

- Address delayed eviction?
- Create fixed buckets (one per prefix) or dynamic buckets?
- Improve performance with asynchronous behavior?
- Handle not just nonresponsive peers, but exceptions that peers generate, etc.?
- Provide reasonable enough documentation to associate the code with the spec?
- Address the difference between local store, cache store, and remote store updates?
- Allow for different protocols?
- Address Sybil attacks?
- Provide detailed unit testing and analysis of the implementation and its performance?

These are some of the issues that we should look for. Even in the initial release of this implementation, I have not addressed all these concerns. Regardless, if you made it this far, I suspect you have a much better understanding of the Kademlia protocol, which also gives you some tools for looking at other P2P protocols.

[19] http://www.maymounkov.org/kademlia