Lecture 25: Security Issues in Structured Peer-to-Peer Networks

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

March 28, 2013
10:27pm

Goals:

- What are peer-to-peer (P2P) overlay networks
- Distributed hash tables (DHT)
- The Chord protocol
- The Pastry protocol
- The Kademlia Protocol
- The BitTorrent File Sharing Protocol
- Security Aspects of Structured DHT-Based P2P Protocols
- Anonymity in Structured P2P Overlay Networks
- An Answer to "Will I be Caught?"

1

CONTENTS

25.1: WHAT ARE PEER-TO-PEER OVERLAY NETWORKS?

- Services in traditional networks (such as the internet) are typically based on the client-server model. Examples include web services provided by your web servers (such as the HTTPD servers) and the browsers that act as clients vis-a-vis the servers. Another common example would be the email servers that are in charge of transporting (sending and receiving) email over the internet and the client email programs running on your personal machine that download email from designated servers.

- Therefore, the traditional services on the internet are based on the concept of central repositories of information; those who wish to see this information must make download requests to the central repositories. This is the same as the relationship between a library and you as a user/member of that library.

- Services in peer-to-peer (P2P) networks are based more on the notion of a book club. All the participants in a P2P network share equally all the information of mutual interest. [Sharing in the

context of a P2P book-club could mean that, for the sake of overall efficiency in storage, a member
participating in a network may choose to store only that chapter that he/she is currently reading.
When he/she decides to look at a chapter that is not currently in his/her own computer, the computer
would know automatically how to fetch it from one of the other members participating in the P2P
network.]

- Modern P2P networks work in a decentralized fashion. That is, there is no machine that acts as a coordinator in the network. All the machines in a P2P network possess the same capability as far as the network is concerned. The machines participating in a P2P network are frequently referred to as **nodes**. [Note that the earliest P2P systems that made this acronym virtually a household word did possess centralized components. **Napster** was the first P2P system that became very popular for sharing music files. Its functioning required a central database for mapping the song titles to the hosts where the songs were actually stored. Then came **BitTorrent** for P2P downloading of large multimedia objects such as movies. The earliest version of BitTorrent also required the notion of a central coordinator that was called the **tracker** which kept track of who had what segments of a large movie file. If you allow for centralized coordinators, constructing a P2P system for file sharing is a relatively easy thing to do. Let's say you are a content provider and you want your files to be downloaded through P2P file sharing. All you have to do is to provide at your web-site a tracker that keeps track of who has requested what file and a client program that folks can download. It would be the job of the client program to talk to the tracker program at your website. As a user, your client program will request a file from the tracker and the tracker would supply your client program with a list of all users currently in possession of the various segments of the file you want (and, at the same time, add you to the list of users who could be in possession of some segments of

the file in question). Your client program would then request the various segments of
the file from their keepers and assemble them back into the file that you were looking
for.]

- As we will see here, the nodes in a P2P network are also **self-organizing**. That is, each new incoming node knows where to place itself in an overall organization of all the participating nodes.

- In addition to being self-organizing, and partly because of it, P2P protocols can allow for such networks to scale up easily.

- Because all nodes participating in a *modern* P2P network operate in an identical fashion and without the help of some sort of a central manager, P2P networks can be characterized as **distributed systems**. [The distributed nature of P2P networks also makes them more **fault tolerant**. That is, the sudden failure of one or more nodes in a network does not bring down the network. When node failures do take place, the rest of the network adapts gracefully. For that reason, P2P systems can also be called **adaptive**.]

- P2P networks are usually overlaid on top of the internet. For that reason, they are also referred to as **overlay networks** or just **overlays**.

• We can therefore talk about routing in the underlying network (usually the internet) and routing in the overlay.

• There are two fundamentally different types of P2P networks: **structured** and **unstructured**. Structured P2P networks generally guarantee that the number of hops required to reach any node in the network is upper-bounded by $O(logN)$ where N is the number of participating nodes; additionally there is the guarantee that a document if present in the network will definitely be reached. [Unstructured P2P networks, as we will see in Lecture 26, do NOT guarantee that a document that was previously stored in the network will be reached.]

• Fundamental to both the structured and the unstructured P2P networks is the concept of the **distributed hash table (DHT)**.

• Here we will only be concerned with the structured P2P networks. Unstructured P2P networks are discussed in Lecture 26.

• In the rest of this lecture, we will first briefly revisit the notion of hash tables. This is to get us ready for introducing the notion of distribution hash tables that come next.

- We will then briefly review three DHT-based P2P protocols: Chord, Pastry and Kademlia. All of these would be referred to as modern implementations of the P2P idea. [As noted previously in this section, while Napster was probably the oldest P2P file-sharing application, it was not a pure P2P system in the modern sense associated with this acronym since it relied on a central database that mapped the song titles to the hosts where the songs were actually stored. This database was made available by a *central index server*. Such a central database would then become a single point of failure for the system. Napster was followed by Gnutella that is fully distributed. Search for resources in early versions of Gnutella was carried out by flooding the network with search requests — a concept that does not scale well as the network grows. Resource location in more modern P2P systems is based on the concept of DHT. Besides Chord, Pastry, and Kademlia, other examples of modern P2P protocols include CAN, Tapestry, Symphony, etc. The very popular BitTorrent, when used in the trackerless mode, also uses DHT for resource location; it is the same DHT as in Kademlia. See Section 25.10 on BitTorrent.]

- Finally, we will talk about the security and anonymity issues related to overlay networks based on DHTs.

25.2: REVISITING THE NOTION OF A HASH TABLE

- As discussed in Lecture 15, the entire idea of hashing is to derive fixed-length records from variable-length documents. Whereas in Lecture 15, our goal was to produce such fixed-length records that could serve as signatures for variable-length documents, here our goal is to create "pointers" that can help us to efficiently locate the documents in a storage system. If we could arrange for matters to be such that each pointer would be unique, we could use the pointers for *constant-time access* to the documents.

- Let's say we want to create a hash table of the phone numbers in a telephone directory so that we can access each name in constant time regardless of the size of the phone book. To keep this explanation simple, assume that all the names consist of single words. That is, the telephone directory looks like

```
avi        333-121-3456
rudy       457-222-8823
stacey     333-456-7890
kim        222-737-8328
mik        234-987-0098
```

Note that each entry in the directory consists of key-value pairs

of the form `<name, phone_number>`.

• To store this directory in a hash table, we first assume that the hash table will be allowed to store only `N` rows. We now design a hash function that directly maps each name to a row index in the table. Let's call this function `f()`. So we want

```
0  <=  f(x)  <=   N-1
```

Obviously, our hope would be that each name would map to a unique row index.

• Let's assume that `N = 128`. We are obviously assuming that our directory will have fewer than 128 names in it.

• Perhaps the most elementary way to hash the names into row indices would be to take the XOR of the ASCII codes associated with the individual letters in the names. Assuming 7 bit ASCII codes, the name **avi** would be hashed into the decimal value 126:

```
    a:      01100001                (hex: 61)
    v:      01110110                (hex: 76)
    i:      01101001                (hex: 69)

  XOR :     01111110                (hex: 7E, dec: 126)
```

So we will assign the row number 126 to the phone entry for **avi**.

• The name "rudy" will get hashed into the decimal 26:

```
    r:      01110010                    (hex: 72)
    u:      01110101                    (hex: 75)
    d:      01100100                    (hex: 64)
    y:      01111001                    (hex: 79)

  XOR :     00011010                    (hex: 1A, dec.: 26)
```

We will assign the row number 26 to the phone entry for **rudy**.

• Similarly, the name "stacey" will get hashed into the decimal 25:

```
    s:      01110011                    (hex: 73)
    t:      01110100                    (hex: 74)
    a:      01100001                    (hex: 61)
    c:      01100011                    (hex: 63)
    e:      01100101                    (hex: 65)
    y:      01111001                    (hex: 79)

  XOR :     00011001                    (hex: 19, dec.: 25)
```

Therefore, we will assign the row indexed 25 to the phone entry for **stacey**.

• By the same token, the name **kim** will get hashed into the decimal value 111:

```
    k:      01101011                    (hex: 6B)
    i:      01101001                    (hex: 69)
    m:      01101101                    (hex: 6D)

  XOR :     01101111                    (hex: 6F, dec: 111)
```

So we will assign the row number 111 to the phone entry for **kim**.

• But note that the name "mik" will ALSO get hashed into the decimal 111:

```
    m:      01101101                (hex: 6D)
    i:      01101001                (hex: 69)
    k:      01101011                (hex: 6B)

  XOR :     01101111                (hex: 6F, dec: 111)
```

• So how do we resolve this **collision**? Both `kim` and `mik` are getting hashed to the same decimal value 111.

• A simpleminded approach to resolving such collisions would be to not just store the phone number at a hash address, but a document record consisting of one or more `<name, phone_number>` pairs, as we show below.

• We can now refer to each hash address as the address of a bucket and store an arbitrary number of `<name, phone_number>` pairs in each bucket. The relationship between the names, the hash indexes, and the buckets could now be shown as

```
    --------------------------------------------------------
    key    |    hash index   |     bucket content
     x     |      f(x)       |
    --------------------------------------------------------

    avi          126              <avi, 3331213456>

    rudy          26              <rudy, 4572228823>
```

```
    kim          111            <kim, 2227378328>    -> <mik, 2349870098>
                                                     |
    mik          111 -------------------------------

    stacey       25             <stacey, 3334567890>

     ...         ...                ....
     ...         ...                ....
    ---------------------------------------------------------
```

- For a more efficient implementation, when we discover a collision, we could move the `<key, phone_number>` pair into the next bucket. If a hash table is sparse, this would make it more likely that each `<key, phone_number>` pair would occupy a single row in the table.

- Instead of XORing the ASCII codes of the characters associated with the keys, we could also base hashing on some of the other simpleminded ideas suggested in Lecture 15. Yet another alternative would be to just add the decimal values associated with characters, calculate this addition modulo a prime number, and use the remainder as the hash index.   [In fact, it was this approach that was suggested originally by Arnold Dumey for hashing in his book "Computers and Automation" way back in 1956. The first person to have coined the term "hash" was the IBM mathematician Hans Luhn in 1953.]

- As you should know already from Lecture 15, the simple-minded methods mentioned above for hashing are NOT cryptographically

secure. Lecture 15 presented the SHA family of cryptographically secure hash functions:

```
SHA-1
SHA-256
SHA-384
SHA-512
```

As mentioned in Lecture 15, the security of SHA-1 was compromised in 2005 when it was demonstrated to be not as secure as originally thought. It nevertheless continues to be used extensively today. From Lecture 15 again, recall that SHA-1 always produces a 160-bit hash of a document.

25.3: DISTRIBUTED HASH TABLES (DHT)

- To illustrate the concept of a distributed hash table (DHT), we will now extend the telephone directory example of Section 25.2.

- Our desire now is to store the telephone directory at a geographically distributed set of machines called nodes. Each node will be characterized by its IP address and the port number it monitors for incoming data lookup queries.

- Let's assume that, at least initially, we have 5 volunteer machines that we have chosen to support our DHT. Let the IP addresses and the port numbers of these 5 nodes be:

```
Node1:        123.45.678.901:6700
Node2:        212.32.678.102:2345
Node3:        86.135.11.1:2378
Node4:        56.135.134.90:7651
Node5:        67.135.134.22:8763
```

- We will now hash each IP address along with the port number into a 7-bit hash by XORing the 7-bit ASCII binary codes associated

with the characters in the node ID strings shown above. This we
can do by using a Perl script such as

```
#!/usr/bin/perl -w
# silly_hash
use strict;
my $str = join ' ', @ARGV;
my @chars = split //, $str;
my $hash = '';
foreach my $char (@chars) {
    $hash ^= "$char";
}
$hash = ord $hash;
print "$hash\n";
```

• When we invoke the above script on the IP address of the first
participating machine, as shown below,

```
silly_hash   123.45.678.901:6700
```

we get the integer value 37. This hash value then becomes the
$nodeID$ of the machine whose IP address and the port number
are given by 123.45.678.901:6700.

• Shown below are the $nodeID$s obtained in this manner for all
five machines participating in our DHT:

```
node IP + port                      nodeID
-------------------                 -------

123.45.678.901:6700                    37

212.32.678.102:2345                    46

86.135.11.1:2378                       18
```

```
        56.135.134.90:7651                                  26

        67.135.134.22:8763                                  30
```

Since we are computing 7-bit hashes, in the form of an integer each $nodeID$ is guaranteed to be between 0 and $2^7 - 1$.

- We can visualize the $nodeID$s for the five participating machines on a circle of all possible hash values, as shown in Figure 1.

- This circle is referred to as the **Identifier Circle**.

- In general, if we compute m-bit hashes, the Identifier Circle will represent integer values between 0 and $2^m - 1$. In our case $m = 7$, so the circle represents integer values between 0 and 127, both ends inclusive. We will start with 0 at the top of the circle and move clockwise; that will make the rightmost point on the circle to stand for the integer value 32. The point at the bottom of the circle will stand for 64; and so on. **We can think of the circle as denoting the modulo 2^m representation of all possible integer values.**

- We will also be interested in distances between any two points on the Identifier Circle. This distance will be the distance measured
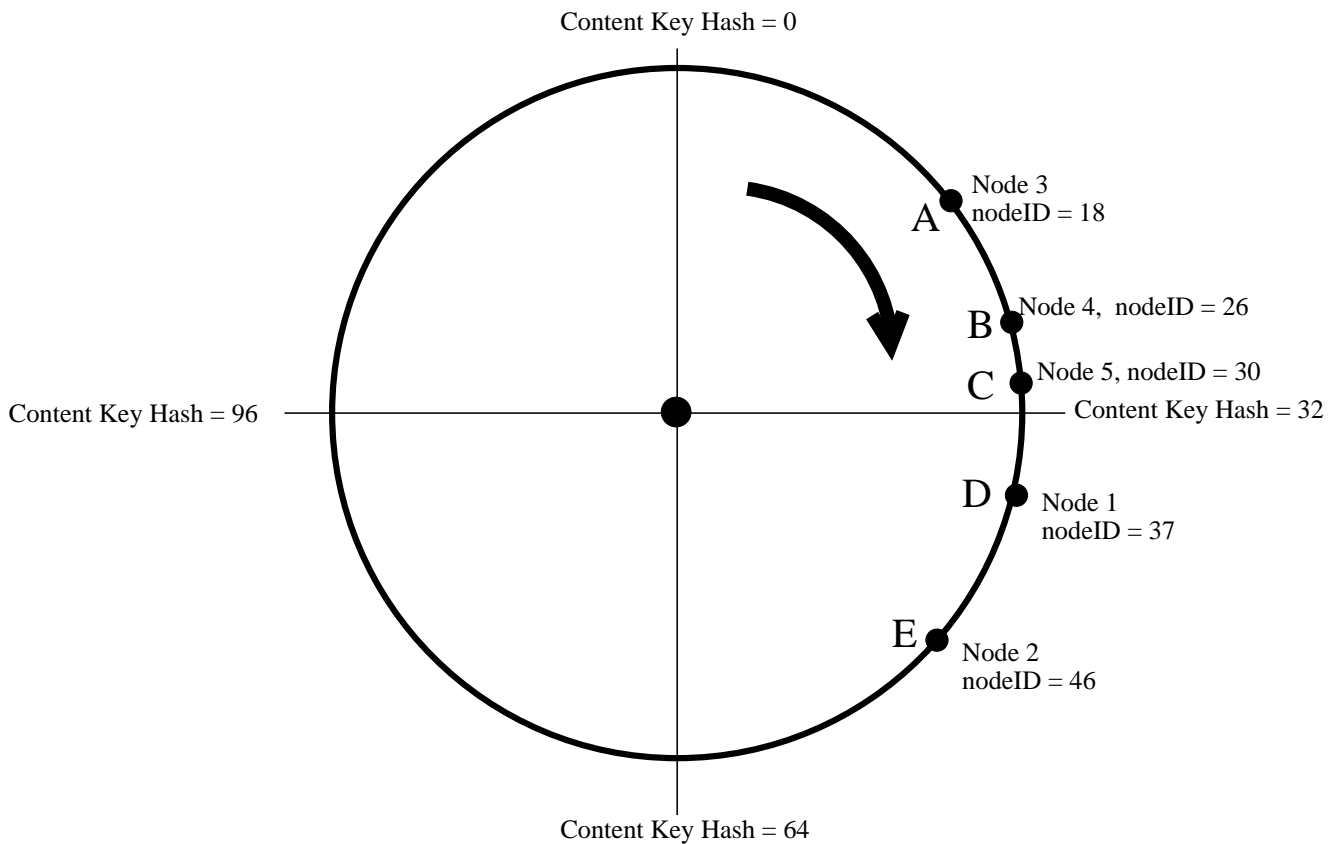
Figure 1: *This figure is from Lecture 25 of "Lecture Notes on Computer and Network Security" by Avi Kak*

clockwise between the two points. For example, the distance between the points A and E is $46 - 18 = 28$. On the other hand, the distance between E and A is $(128 + 18) - 46 = 100$.

• Now we are ready to get down to our main business, which consists of storing the telephone directory shown in Section 25.2 in a distributed manner in the five nodes of the DHT. Our overall approach will be to compute the 7-bit hash for each name in the telephone directory and locate that hash value on the Identifier Circle of Figure 1. We will refer to the hashes of the names in the telephone directory as **content keys** or just as **keys**. Recall that each directory entry can be considered to be a `<key,value>` pair with the name of the individual serving as the key and the phone number as the value. [Although we are using the telephone directory as an example of **key-indexed storage**, the web is an even more powerful example of the same. In the web, the URLs are the keys and, for each key, the web page at that URL the value or the data.]

• We must next figure out the point on the Identifier Circle where a given content key belongs. What we need is a policy regarding how to assign various segments of the Identifier Circle to each of the network nodes already placed on the circle (See Figure 1 that shows five live nodes already situated on the circle).

- We could, for example, use the policy shown in Figure 2. This policy says that all content keys between any two consecutive nodes on the Identifier Circle will become the responsibility of the network node at the end of the circle segment. More precisely, all the keys that are between A's *nodeID* plus 1 and B's *nodeID*, inclusive of both ends, are assigned to B. Similarly with the other segments of the Identifier Circle in the figure.

- The policy shown in Figure 2 can be implemented by writing a function that could be called `lookup(key)`. Given any point on the Identifier Circle that corresponds to a key, this function is supposed to return the IP address of the participating node that is responsible for that key. We can think of `lookup(key)` as a part of a database client program that could run on any machine authorized to access the DHT. **But note that `lookup(key)` must possess a distributed implementation.** This could be done by each node in the overlay network maintaining a **successor pointer** to the next node on the Identifier Circle. So when a query is received by a node concerning a particular key value *key*, if the value of *key* exceeds the *nodeID* of the node, it would forward the query to the successor node. More efficient distributed implementations for `lookup(key)` will be presented when we discuss the Chord and the Pastry protocols for P2P.

- With the key-to-*nodeID* assignment policy shown in Figure 2,
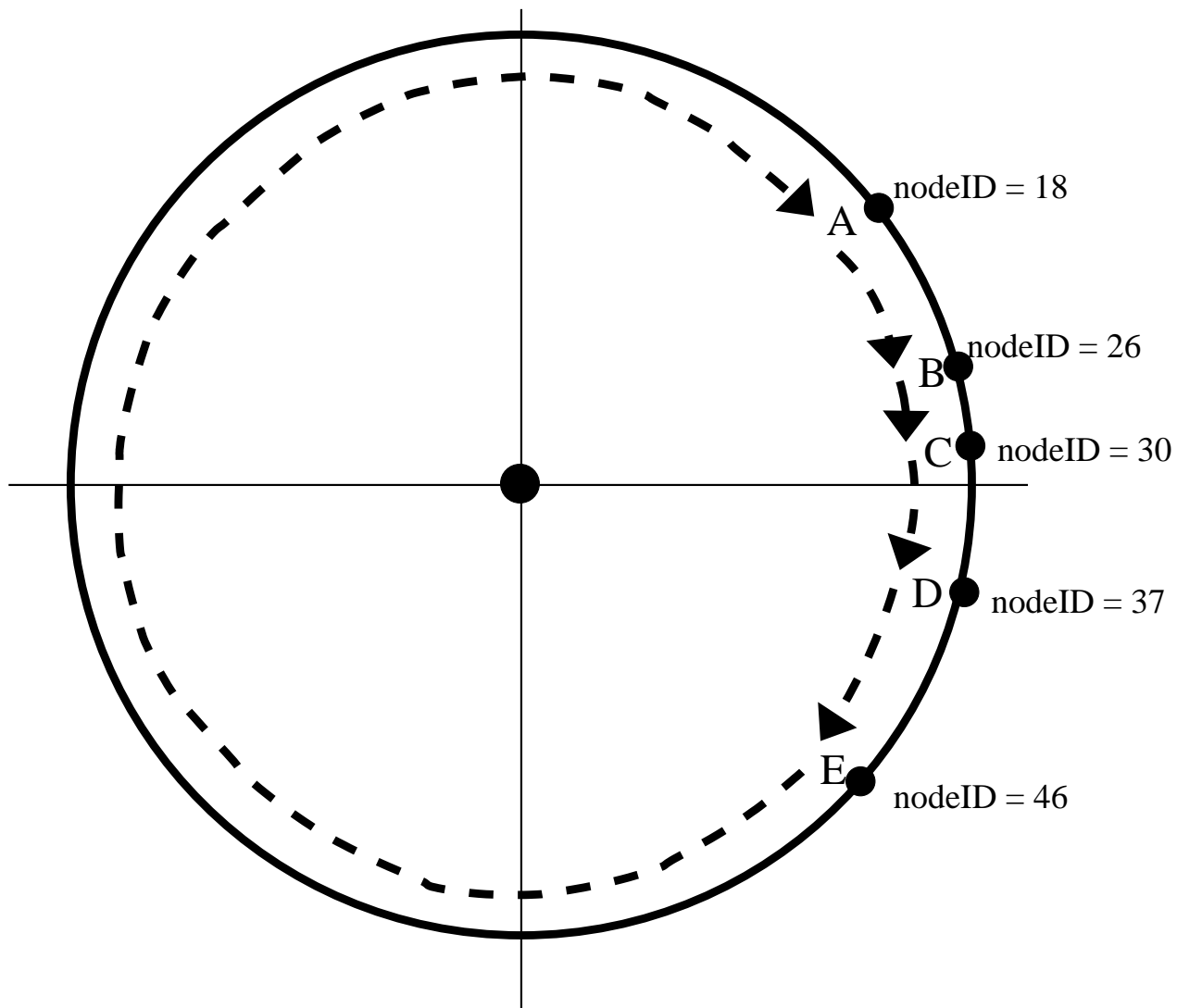
All content−key hash values between A+1 and B assigned to the node at B
All content−key hash values between B+1 and C assigned to the node at C
All content−key hash values between C+1 and D assigned to the node at D
All content−key hash values between D+1 and E assigned to the node at E
All content−key hash values between E+1 and A assigned to the node at A

Figure 2: *This figure is from Lecture 25 of "Lecture Notes on Computer and Network Security" by Avi Kak*

we are now ready to store in our DHT the telephone directory
that was presented earlier on Slide 6. We apply the `silly_hash`
Perl script to each name in the telephone directory of Slide 6
to obtain the hash values shown below in the right column. As
stated earlier, these will be called the content keys or just the
keys.

```
name in directory              content key
-----------------              -----------------

     avi                            126

     rudy                           26

     stacey                         25

     kim                            111

     mik                            111
```

- We now locate these keys on the Identifier Circle, as shown in
  Figure 3. With the key-to-node assignment policy of Figure 2,
  we must assign the entries for "Rudy" and "Stacey" to Node B,
  and for "Kim", "Mik", and "Avi" to Node A in the Identifier
  Circle of Figure 3.

- The above scheme for distributed storage of information would
  work reasonably well if we had a fixed set of nodes participating
  in a P2P network. For a fixed set of nodes, a DHT would need to
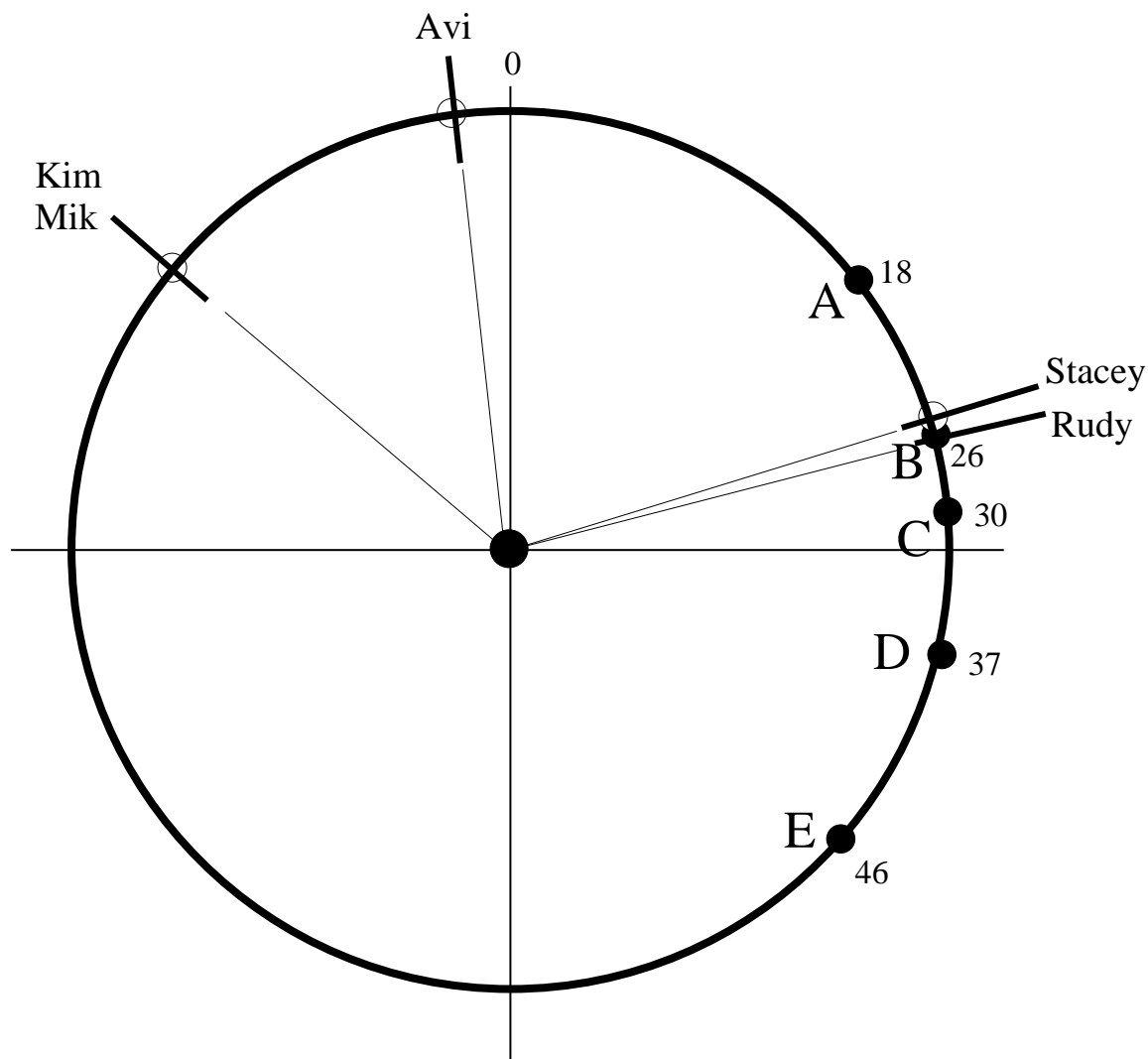 support just one operation `lookup(key)` that should return the

Figure 3: *This figure is from Lecture 25 of "Lecture Notes on Computer and Network Security" by Avi Kak*

IP address of the network node that owns the argument content key.

- In actual practice, P2P networks tend to be highly dynamic. Nodes can join and leave at will. So a protocol for data lookup in a P2P network must allow for this sort of churn. How that is accomplished depends on which P2P protocol you use.

- To mention it summarily here, a real-life P2P protocol must provide facilities for:

  - Mapping content keys to network nodes while observing load balancing considerations.

  - Each node must be able to forward a given content key to a node whose ID hash is closer to the content key.

  - The nodes must be able to build **routing tables** adaptively as new nodes join and existing nodes leave. As you will see later, routing tables are used to speed up the search for the node that is closest to a given content key and to facilitate recovery from node failures.

- Before we present examples of protocols that possess the above-mentioned properties, we will next talk briefly about a property

of DHTs that is called **consistent hashing**.

25.4: CONSISTENT HASHING

- Our DHT explanation so far would probably suffice for construct-
  ing a distributed database (albeit one that is not very efficient
  with regard to key lookup).

- Our previous DHT explanation left out one critical question: How
  do we let new nodes join the network and existing nodes leave it
  at their own pleasure? A related question would be: How do we
  make sure that our distributed database can handle node failures?

- The question regarding new nodes joining in and old nodes leav-
  ing has to be examined from the perspective of the extent to
  which the content keys must be reassigned to the various nodes.
  The notion of **consistent hashing** addresses this issue.

- We refer to a DHT scheme as consistent hashing if the insertion
  of a new node into the P2P overlay affects only the information
  stored at the machines whose *nodeID*s are closest to the new

node joining the overlay. For consistent hashing, it must also be true that the removal of an existing node should affect only the nodes that are still in the overlay and whose $nodeIDs$ are closest to the departing node on the Identifier Circle.

- Consistent hashing is a highly desirable property of DHT schemes because it minimizes the reorganization of the stored data in the presence of high churn. **Churn** refers to nodes joining or leaving a P2P overlay at will.

- We will next review two P2P protocols, Chord and Pastry, and see how these practical issues are dealt with in these protocols.

25.5: THE CHORD PROTOCOL

- The Chord protocol was created by Stoica, Morris, Karger, Kaashoek, and Balakrishnan. [Proc. ACM SIGCOMM 2001]

- The Chord protocol uses for node identities the Identifier Circle shown in Figure 1. The $nodeID$ for each node is typically calculated by hashing its IP address using the SHA-1 algorithm (see Lecture 15). As a result, each $nodeID$ is a 160-bit integer and we have a maximum of 2^{160} points on the Identifier Circle of Figure 1. In keeping with our earlier explanation of how DHT works, the content keys are also calculated with the same SHA-1 algorithm. The goal is to create a distributed database in which a content document is stored at a node whose $nodeID$ is closest to the content key going clockwise on the Identifier Circle.

- Each physical node participating in a Chord overlay network maintains a **successor pointer** and a **predecessor pointer**. [Actually, as we will see later, each node maintains a list of a certain number of nearest successors for greater efficiency in content location and to facilitate recovery from node failures.]

• Note that the notion of a *successor pointer* applies to a live node in the overlay. For a given live node on the Identifier Circle, the successor pointer consists of the $nodeID$ and the IP address of the live node that is *next* on the Identifier Circle.

• We will also talk about a function $successor(key)$. We want this function to return the $nodeID$ of the next live node on the Identifier Circle after the point that corresponds to the content key *key*. That is, the function $successor()$ invoked on the content key *key* should return the Identifier Circle location of the live node responsible for the key *key*.

• Strictly speaking, the Chord protocol needs to know only the *successor pointer* at each live node for the protocol to work correctly. However, in order to speed up the process of successor location for an arbitrary content key on the Identifier Circle, each live node additionally maintains a **routing table** with at most m entries in it where m is the number of bits used in representing $nodeID$s and the content keys. With SHA-1 as the algorithm that calculates the $nodeID$s and the content keys, $m = 160$. So the routing table at each node will have at most 160 rows in it.

• The first entry in the routing table at the node whose $nodeID$ is n is the IP address of the successor node to the *hypothetical*

key $n+1$ on the Identifier Circle; the next entry the IP address of
the successor node to the hypothetical key $n+2$ on the Identifier
Circle; the entry below that the IP address of the successor node
to the hypothetical key $n+2^2$ on the Identifier Circle; and so on.

- So, in general, the ith row in the routing table contains the IP
  address of the successor node to the hypothetical key $n + 2^{i-1}$
  for $1 \leq i \leq m$ where m is the number of bits used to represent
  the $nodeIDs$ and the keys. So if the entry in the ith row of the
  routing table is the IP address of a node whose $nodeID$ is s_i, we
  can write

$$s_i \quad = \quad successor(n + 2^{i-1}) \qquad\qquad 1 \leq i \leq m - 1$$

  where the addition in the argument to $successor()$ is computed
  modulo 2^m.

- Whereas m is the maximum number of rows in the routing table
  at each node, for obvious reasons the number of successors listed
  will not exceed N, the actual number nodes participating in the
  overlay. So if $N < m$, which is not an unlikely scenario for a small
  overlay, several of the entries in the routing table may point to
  the same successor node.

- With the above construction of the routing table, each participat-

ing node has a detailed "perception" of the nodes that are ahead of its own position but in its own vicinity on the Identifier Circle. This perception becomes increasingly coarse — coarser by halves, to be precise — for nodes that are farther out on the Identifier Circle.

• This is how the routing table is used to handle a query for a content key k: This query can be submitted to any live node on the Identifier Circle. Let's say that the query goes to a node whose $nodeID$ is n. If k were to equal n, or $n + 1$, or $n + 2$, we would directly find in the routing table the successor nodes for that content key. For any other value of k, the routing table is queried for an entry whose $nodeID$ j immediately *precedes* k. For obvious reasons, the node that is a successor to j is more likely to own the content key k than the node n was. Through recursive lookups of the routing tables in this manner, each contacted node n is bound to get closer and closer to the node that actually owns the key k.

• The developers of Chord have theoretically established that the number of nodes that must be contacted to find a successor in an N-node network is $O(logN)$ with high probability.

• Let's now talk about how a new node joins the network.

• To simplify joining (or leaving) the overlay, each participating node in a Chord network maintains what is known as a **predecessor pointer**, which is the $nodeID$ and the IP address of the node immediately preceding the node in question on the Identifier Circle. So whereas the successor pointers we mentioned earlier allow a clockwise traversal of the Identifier Circle, the predecessor pointers would allow a counterclockwise traversal of the circle.

• A new node that wants to join a Chord overlay computes its $nodeID$ and contacts any of the existing nodes with that information. Assume that the $nodeID$ of the new node is n. Also assume that the $nodeID$ of the node contacted by the new node is n'. The new node n queries n' as to what its (meaning, n's) *successor* is in the Identifier Circle. Let this successor be n_s. The new node then links itself into the Identifier Circle by making n_s its immediate successor and making n_s's predecessor its own predecessor. The node n also fills up its routing table by using the entries in the routing table for n_s. Subsequently, n_s updates its own routing table and makes n its immediate predecessor. Finally, the entries in the routing tables of all the nodes are updated taking into account the new node. [The developers of Chord have shown that, with high probability, the number of nodes that need to update their routing tables is $O(logN)$ where, as before, N is the number of nodes in the overlay.] Finally, the content keys that should be assigned to the new node are transferred from n_s to n.

- When a node whose $nodeID$ is n leaves the network, its predecessor in the Identifier Circle must update its successor pointer to what was n's successor. By the same token, this latter node must update its predecessor pointer to point to what was n's predecessor. As a last step, all of the content keys that were assigned to the departing node must now be reassigned to what was n's successor.

- In order to deal with random joins and departure of nodes, Chord runs a special high-level program, $stabilize()$, at every node every 30 seconds. When a newly joined node n runs its $stabilize()$, it is the stabilizer's job to make sure n's successor has a correct predecessor.

- Potential loss of content (that is, the data associated with the content keys) stored at a node that may have failed or departed without notification is dealt with by storing a list of immediate successor nodes at each live node and replicating content between multiple immediate successors.

25.6: NODE PROXIMITY ISSUES IN ROUTING WITH DHTs

- The basic Chord protocol as described in Section 25.5 suffers from an interesting "shortcoming": A good hashing algorithm — and SHA-1 is a very good hashing algorithm despite the security concerns raised recently — will distribute the $nodeID$ values all over the Identifier Circle even when the IP addresses of the nodes are closely related. In what follows, we will explain why this could degrade the performance of a Chord overlay network.

- Say that you have a dozen machines participating in a Chord overlay; half of these are on the local network in your lab in USA and the other half in another lab somewhere in India. Since SHA-1 will create a large change in the hash values for even very small changes in the IP addresses associated with the machines, when you locate the 12 nodes on the Identifier Circle of Figure 1, the nodes would be situated in a more or less random order as you walk around the circle. That is, you will not see any clustering of the nodes corresponding to the six machines in the US and the six machines in India.

• So when an application program seeks the overlay node that is responsible for a given content key, in all likelihood that query will make multiple hops around the globe even when the overlay node of interest is sitting right next to the computer running the application program.

• This problem arises because the basic Chord protocol does not take into account any proximity between the nodes in deciding how to route the queries. **By proximity between two nodes in the overlay we could mean the number of hops between the nodes in the network that underlies the overlay.**

• The next P2P protocol we present, Pastry, is more aware of proximity between the nodes. Of all the nodes that are candidates for receiving a query, it will try to choose one that is most proximal to the one where a query is originating.

25.7: THE PASTRY PROTOCOL

• The Pastry protocol was created by Rowstron and Druschel.
[Proc. 18th IFIP/ACM Conference on Distributed Systems Platforms, 2001]

• Pastry, like Chord, creates a self-organizing overlay network of
nodes. As in Chord, each participating node is assigned a $nodeID$
by possibly hashing its IP address and port number.

• Pastry uses a 128-bit hash for $nodeID$s and for content keys. So,
on the Identifier Circle (see Figure 1), the numeric address of a
node is an unsigned integer between 0 and $2^{128} - 1$.

• When deciding at which node to store a message, Pastry uses the
same basic rule as Chord: A message is delivered to the node
whose $nodeID$ is closest to message key. But Pastry gets to that
final node in a manner that is different from Chord.

• What distinguishes Pastry from Chord is that the former takes into account *network locality* by using a **proximity metric**.

• The proximity metric could be the number of IP routing hops in the underlying physical network. **It is the higher-level application program that is supposed to supply the proximity metric.** The application program could, for example, use a utility such as `traceroute` to estimate the number of hops between any two nodes. By taking into account network locality through the proximity metric, Pastry tries to minimize the distance traveled by messages.

• With regard to how messages are routed, another difference between Chord and Pastry is how a content key is compared with the *nodeID*s in order to decide which node to forward a query to. The comparison of the hash values is carried out using base-b digits. For example, with base-16 digits, Pastry would compare the hex digits of a content key with the hex digits of a *nodeID*. This comparison looks for the **common prefix** between the two.

• Pastry makes a routing decision on the basis of the length of the above-mentioned prefix; the length of the prefix shared with the current node's *nodeID* is compared with the length of the

prefix shared with the next node's $nodeID$. **The goal is to make the shared prefix longer with each routing step.** However, if that is not possible, the goal is to select a node whose $nodeID$ is numerically closer to the content key but, at the same time, whose prefix shared with the key is no shorter than what is the case at the current node.

- With the routing scheme described above, the number of correct digits in the $nodeID$ of the next node chosen as a query is forwarded will always either increase or stay the same. If it stays the same, the numerical distance between the $nodeID$ of the node chosen and the content key will decrease. **Therefore, the routing protocol must converge.**

- The above-mentioned routing decisions are made with the help of a routing table maintained at every node. If b-bit digits are used for comparing a $nodeID$ with a key, then the routing table consists of $128/2^b$ rows and 2^b columns. Since typically $b = 4$, the routing table for a typical Pastry network node will have 8 rows and 16 columns.

- Assuming $b = 4$, Figure 4 shows the order in which the IP addresses would be stored in the routing table at a node whose $nodeID$ is 3a294f1b. Recall that the comparison between the

nodID's in this case is carried out using hex digits. Each row of the table orders the IPs according to the *nodeID*s associated with them. In the 0th row, the entries are ordered in increasing order of the first digit in the *nodeID*s. Since there 16 possible values for the first digit, we will have 16 entries in the first row. Note the empty cell in the first row of the routing table — this cell is empty because it corresponds to all possible nodes in the overlay whose *nodeID*'s begin with the prefix digit 3. **All such nodes in the overlay are represented by the rest of the table.** Along the same lines, note the empty cell in the second row of the table. This cell is empty because it corresponds to all possible nodes in the overlay that begin with the prefix 3a. **All such nodes in the overlay are represented by the rest of the table below the second row**, and so on.

- Note that in general there will not exist an IP entry in every non-empty cell of the routing table shown in Figure 4. The table shows only the order in which the IP addresses of the nodes would be stored in the routing table if such nodes are indeed active in the overlay. If there does not exist in the overlay a node corresponding to any of the non-empty cells in Figure 4, then that cell would be empty of an IP address.

- In case the reader is wondering as to how the routing table gets filled, we need to talk about Pastry's `join` operation that allows

Routing Table for a Node of nodeID   =   *3a294f1b*

x   =   **arbitrary suffix**

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Row 0	0x	1x	2x		4x	5x	6x	7x	8x	9x	ax	bx	cx	dx	ex	fx
Row 1	30x	31x	32x	33x	34x	35x	36x	37x	38x	39x		3bx	3cx	3dx	3ex	3fx
Row 2	3a0x	3a1x		3a3x	3a4x	3a5x	3a6x	3a7x	3a8x	3a9x	3aax	3abx	3acx	3adx	3aex	3afx
Row 3	3a20x	3a21x	3a22x	3a23x	3a24x	3a25x	3a26x	3a27x	3a28x		3a2ax	3a2bx	3a2cx	3a2dx	3a2ex	3a2fx
Row 4	3a290x	3a291x	3a292x	3a293x		3a295x	3a296x	3a297x	3a298x	3a299x	3a29ax	3a29bx	3a29cx	3a29dx	3a29ex	3a29fx
Row 5	3a2940x	3a2941x	3a2942x	3a2943x	3a2944x	3a2945x	3a2946x	3a2947x	3a2948x	3a2949x	3a294ax	3a294bx	3a294cx	3a294dx	3a294ex	
Row 6	⋮		⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Row 7	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮		⋮	⋮	⋮	⋮	⋮

This cell is left empty because it corresponds to all nodes with the prefix 3, the first digit of the nodeID 3a294f1b. The rest of the table below is for all such nodes.

This cell is left empty because it corresponds to all nodes with with the prefix 3a, the first two digits of 3a294f1b. The rest of the table below is for all such nodes.

This cell left empty because it corresponds to all nodes with the prefix 3a2, the first three digits of 3a294f1b. The rest of the table below is for all such nodes.

This cell is left empty because it corresponds to all nodes with the prefix 3a29, the first four digits of 3a294f1b. The rest of the table below is for all such nodes.

This cell is left empty because it corresponds to all nodes with the prefix 3a294, the first five digits of the nodeID 3a294f1b. The rest of the table below is for such nodes.

This cell left is empty because it corresponds to all nodes with the prefix 3a294f, the first six digits of the nodeID 3a294f1b. The rest of the table below is for such nodes.

Figure 4: *This figure is from Lecture 25 of "Lecture Notes on Computer and Network Security" by Avi Kak*

a new node to join the overlay.

- When a new node wishes to join the overlay, it sends its 128-bit $nodeID$ to a node that is currently active in the overlay and that hopefully is close to the new node in terms of the proximity metric used. Let's denote the new node's $nodeID$ by X and the currently active node that X first contacts as A. The node A then sends a join message to the rest of the nodes to discover the node whose $nodeID$ is closest to X. This message propagates in the overlay like any other query message, except for the fact that any nodes encountered along the way send their routing tables back to X. Based on the information received, and possibly on the additional information queried from other nodes, node X initializes its own state (and that includes its routing table).

- In addition to the routing table, each node also maintains a **leaf set** that consist of a maximum of l nodes whose $nodeID$s are numerically closest to that of the present node. Of these, $l/2$ are the nodes whose $nodeID$s are larger than that of the current node and $l/2$ nodes those whose $nodeID$s are smaller than that of the current node. The value of l, constant for all the nodes in a network, is typically $8 * log_{2^b}N$ where N is the total number of nodes in the overlay. Since $b = 4$ commonly, that would l typically equal to $8 * log_{16}N$. Nodes in the leaf set are used to seek out a node closest to the current node, in accordance with

the routing rules mentioned earlier. Nodes in the leaf set are also
used for storing copies of the content information; this is done to
make sure that the information is not lost when a node fails or
otherwise leaves the network.

- If the prefix-based routing rules described earlier do not yield a
  suitable target node from the routing table and if the leaf set
  also does not yield one, then the current node or its immediate
  neighbor is the query's final destination.

- Pastry's prefix-based routing results in the number of routing
  hops being bounded by approximately $log_{16}N$ where N is the
  number of nodes in the overlay and when base-16 digits are used
  for comparing keys with $nodeIDs$.

25.8: THE KADEMLIA PROTOCOL

- Kademlia was developed by Maymounkov and Mazieres. [IPTPS02 2002].

- Kademlia is important because its DHT is employed by the very popular BitTorrent protocol (for downloading music and movies) when it is used in a trackerless mode. A brief review of BitTorrent is presented in Section 25.10.

- Kademlia uses the same identifier space as Chord (Figure 1). Each node wishing to join a Kademlia overlay typically uses SHA-1 to generate a 160-bit value for its *nodeID*. The key values are also generated in the same manner as in Chord — by applying SHA-1 to the data that needs to be stored. Again as in Chord and Pastry, data for a given content key is stored at a node whose *nodeID* is closest to the key.

- To understand routing in Kademlia, you have to understand how this protocol measures the "distance" between two points on the

Identifier Circle of Figure 1. Since a new idea is sometimes best understood by comparison with an older version of the same idea, let's first review how Chord and Pastry measure distances in the identifier space.

• As the reader will recall, Chord measures the distance from a point A to a point B on the Identifier Circle of Figure 1 by going **clockwise** from A to B and subtracting (modulo 2^{160}) the integer value of A from the integer value of B. This notion of distance between two points in the identifier space is asymmetric with respect to the points. On the other hand, as explained earlier, Pastry uses two separate methods for computing the distance between two points in the identifier space: In the first method, Pastry sets the distance on the basis of the shared prefixes in the base-b representations of the two points and, in the second method, the distance is computed in the same way as in Chord. Whereas Pastry's first method for computing the distance between two points is symmetric with respect to the points, the second method, being the same as in Chord, is asymmetric.

• Compared to Chord and Pastry, Kademlia measures the distance between any two points A and B on the Identifier Circle of Figure 1 by taking the XOR of the two bit patterns. [If $d(A, B)$ denotes the XOR of the bit patterns corresponding to the points A and B on the Identifier Circle, it can be shown easily that d is a metric: $d(A, B) = 0$ if an only if $A = B$; $d(A, B) \geq 0$ for A and B;

$d(A, B) = d(B, A)$; and, finally, $d(A, B) + d(B, C) \geq d(A, C)$. The triangle inequality follows from the fact that $d(A, C) = d(A, B) \oplus d(B, C)$ and the fact that $\forall A \geq 0, \forall B \geq 0 \quad A + B \geq A \oplus B$.]

- Since the XOR metric is symmetric, a node can receive a query from a node in its own routing table. [For the sake of a comparison, the metric used in Chord for comparing two values of $nodeID$ is asymmetric, as mentioned previously. Since Chord measures distances in the clockwise direction only on the Identifier Circle, a node A can be close to B but B may not be close to A.] The symmetry in the metric used to measure the distances in the identifier space allows a Kademlia node to send queries to all nodes in its vicinity.

- In Kademlia, the routing table at each node consists of a maximum of 160 separate lists when a 160-bit representation is used in the identifier space. The list for each $0 \leq i \leq 160$ at a node consists of the connection information for all the nodes that are at a distance between 2^i and 2^{i+1} from itself. The connection information on each destination node consists of the IP address, the UDP port, and the $nodeID$ value.

- The list of the nodes for each i is referred to as a k-bucket. The reason for k in the name k-bucket will become clear shortly.

- Each k-bucket is kept sorted by the time last seen, with the

least recently "seen" node at the the head of the list and the
most recently "seen" node at the tail. It will soon become clear
as to what is meant by "seen."

• Since the distance between 2^i and 2^{i+1} can be very small for small
  i, it is possible for the k-buckets for small i to be empty. For
  large i, we keep a maximum of k nodes in the k-bucket, with k
  being typically set to 20. Kademlia refers to k as the system-wide
  replication parameter.

• When node A receives a message (query or reply) from node
  B, A updates the appropriate k-bucket depending on the dis-
  tance between $nodeID$ values for A and B. If B is already in
  the k-bucket, it is moved to the tail of the list. If B is not in
  the k-bucket and the list is not full, B is still moved to the tail
  of the list. If B is not in the k-bucket and the list is full, the
  node at the head of the list — the least recently seen node —
  is pinged. If the response to the ping times out, it is removed
  from the k-bucket and B inserted at the tail. However, if there
  is a response to the ping, the pinged node is moved to the tail
  of the list and B simply ignored with regard to its insertion in
  the routing table. The authors of Kademlia refer to this as the
  least-recently seen eviction policy.

• When a `<key,value>` is stored in the DHT, for data replication
purposes it is stored in the k nodes that are nearest to that key.
This is the same as what happens in Chord and Pastry. However,
the procedure used to discover the k nodes closest to a key is
different in Kademlia.

• The search for the k closest nodes to a given key begins by select-
ing α "contacts" from the closest `k-bucket` of any node in the
overlay. Let A be the node where we search for the k closest nodes
to a given key. We therefore start at A with the `k-bucket` that is
closest to the key in question. If there are fewer than α nodes in
that bucket, the node A selects nodes from the `k-bucket` that is
next closest to the key and so on until a pool of α nodes has been
constructed. This list of α nodes is referred to as the **shortlist**
for the search. The node A then sends out parallel asynchronous
requests to all the nodes in the shortlist. If any of the targeted
nodes fails to reply, the node A removes it from the shortlist.
From the replies that are received, the node A reconstitutes with
the k nodes closest to the key in question. This process continues
iteratively at the node A until no further nodes are dropped from
the shortlist.

• The above procedure for finding the k nodes closest to a given
key is referred to as $node\_lookup(key)$.

- One big advantage of sending out parallel asynchronous queries in $node\_lookup(key)$ is that timeout delays from failed nodes are minimized. Note that α is a system-wide concurrency parameter, usually set to 3.

- Many of the operations in Kademlia are based on $node\_lookup(key)$.

- When a new node wishes to join a Kademlia overlay, it first computes its own $nodeID$ and then inserts the contact triple (IP address, UDP port, and the $nodeID$ number) of some known active node in the overlay into the appropriate bucket as its first contact. The new node invokes $node\_lookup(key)$ with the argument key set to its own $nodeID$. This step populates the k-buckets of the currently active nodes that are contacted by $node\_lookup(key)$ with the contact triple of the new node. After this, the new node refreshes its k-buckets by calling $node\_lookup(key)$ using values for key set randomly to a point in the intervals covered by the k-buckets.

- Finally, there exists a Python implementation of Kademlia — it is called Khashmir — that is used in BitTorrent. Other Python implementations of Kademlia include SharkyPy and Entangled.

25.9: SOME OTHER DHT-BASED P2P PROTOCOLS AND A COMPARISON OF THE PROTOCOLS

• The other DHT-based P2P protocols that have also received much attention include CAN (Content Addressable Network), Tapestry, RSG (Rainbow Skip Graph), Viceroy, and so on.

• Tapestry's routing algorithm is similar to Pastry's. Therefore, like Pastry, it can include node proximity in its criterion for selecting entries for the routing table at each node. Tapestry is also based on the one-dimensional circular *nodeID* space used in Chord, Pastry, Kademlia, etc.

• Whereas Chord, Pastry, Kademlia, etc., route messages in a one-dimensional circular *nodeID* space, CAN routes messages in a d-dimensional space. Each node maintains a routing table with $O(d)$ entries in it. The entries in the routing table refer to the node's neighbors in the d-dimensional space. Like Chord, CAN is not able to take into account node proximities.

• Shown next are two tables, the first lists some performance metrics for comparing DHT-based P2P protocols, and the second a comparison of some of the DHT-based P2P protocols with respect to the metrics.

(1)	Messages required for each key lookup
(2)	Messages required for each store lookup
(3)	Messages needed to integrate a new peer
(4)	Messages needed to manage a peer leaving
(5)	Number of connections maintained per peer
(6)	Topology can be adjusted to minimize per-hop latency (yes/no)
(7)	Connections are symmetric or asymmetric

Table 1: This table is reproduced from "Routing in the Dark: Pitch Black" by Nathan Evans, Chris GauthierDickey, and Christian Grothoff

	Chord	Pastry	Kademlia	CAN	RSG
(1)	$O(log\ N)$	$O(log\ N)$	$O(log\ N)$	$O(N^{-d})$	$O(log\ N)$
(2)	$O(log\ N)$	$O(log\ N)$	$O(log\ N)$	$O(N^{-d})$	$O(log\ N)$
(3)	$O(log^2\ N)$	$O(log\ N)$	$O(log\ N)$	$O(N + N^{-d})$	$O(log\ N)$
(4)	$O(log^2\ N)$	$O(1)$	$O(1)$	$O(d)$	$O(log\ N)$
(5)	$O(log\ N)$	$O(log\ N)$	$O(log\ N)$	$O(d)$	$O(1)$
(6)	no	yes	yes	yes	no
(7)	asymmetric	asymmetric	symmetric	symmetric	asymmetric

Table 2: This table is reproduced from "Routing in the Dark: Pitch Black" by Nathan Evans, Chris GauthierDickey, and Christian Grothoff

25.10: THE BitTorrent PROTOCOL

• Because this protocol has become extremely popular for fast downloads of large video and movie files on a peer-to-peer basis and because, in some of its versions, the protocol uses the Kademlia DHT, it is appropriate to review it briefly here.

• There is no official specification of the BitTorrent protocol that was originally developed by Bram Cohen. Bram Cohen created a BitTorrent client that is now referred to as the **mainline** client. **The mainline client serves as a reference for the protocol.** [A casual reader might think that if we have BitTorrent clients, we must also have BitTorrent servers. Strictly speaking, there is no such thing as a BitTorrent server. Therefore, all machines that run the BitTorrent software are clients. A BitTorrent client does both the downloading and the uploading of the different pieces of a file. BitTorrent creates a peer-to-peer network for exchanging the different pieces of a large file. However, note that some folks refer to the node on which the tracker is running as severs. (As you we will see later in this section, a tracker stores the content key for each media object that is available for P2P download and, for each content key, a list of IP/port addresses of the peers currently distributing the content.) Using a DHT, however, it is possible to run BitTorrent in a pure P2P trackerless mode. You could say that with DHT, the tracker becomes distributed over all the nodes participating in the DHT.]

- BitTorrent breaks a large file into smaller pieces called **blocks** that can subsequently be downloaded by clients by interacting with other clients possessing different pieces of the same file. A client that has collected all the blocks is called a **seeder**. And a client that is still collecting the blocks is called a **leecher**. A block is typically 250 kilobytes in size.

- Let's say you want to make a file available for a BitTorrent download by others. The first thing you do is to use the BitTorrent software to create a *torrent* file; this is a file whose name ends in the suffix ".torrent". The torrent file contains the following sections:

  - – an "announce" section that mentions the URL of the tracker.

  - – an "info" section that mentions the block size used and SHA-1 hash for each block

- The tracker associates with a `.torrent` file the currently list of peers, these being nodes that currently possess different pieces of the file. This list is updated by the tracker as new nodes join a **swarm** and the old nodes leave. By definition, a **swarm** is the current set of peers engaged in exchanging different pieces of a file.

- Someone wishing to download a large file starts out by downloading the small `.torrent` file related to the desired download. The `.torrent` file tells the BitTorrent client where the tracker is located and the tracker informs the client what other peers are currently active in the swarm. [A BitTorrent client queries a tracker with a SHA-1 hash of the `.torrent` file. This hash serves as the content key for the media object that the user wants to download through BitTorrent. When querying a tracker, a client also subscribes to the tracker its IP address and the port number. The tracker returns to the client the IP addresses and the associated ports for all of the hosts that are currently in the same swarm. Therefore, the BitTorrent client running on your machine can see the IP addresses of all of the folks who are downloading the same media content. Since anyone can join a swarm for downloading any content whatsoever, what this means is that there is no anonymity at all for the downloaders of media content through BitTorrent. This lack of anonymity is further exacerbated by the fact that the communication between the tracker and the client is in plaintext. Therefore, anyone monitoring the traffic between a client and a tracker would be able to get information on the participants in a swarm even without having to join the swarm. Note that the communication between a client and tracker may be either through TCP or UDP. In either case, the security ramifications are the same.] If only the initial seeder for the file is available, the client connects with the seeder and starts downloading the different file pieces. As other clients join in by checking in with the tracker, thus creating a swarm, the clients start trading pieces with one another.

- What I have described so far is the "traditional" way of using BitTorrent for downloading large files. This approach suffers from the flaw that the service provider that serves as a clearing house

for the `.torrent` files becomes a single point of failure for content delivery. A large service provider would obviously construct an index of all the `.torrent` files it can make available to the BitTorrent clients.

- A second shortcoming of the "traditional" approach is the heavy burden it places on the trackers. The world's largest repository of `.torrent` files, http://thepiratebay.org, used to maintain eight BitTorrent trackers for all the incoming traffic for P2P downloads. A user's BitTorrent client would first download a `.torrent` file from the web site and then approach one of the eight trackers with the content ID (SHA-1 hash of the the torrent file) in order to join the swarm related to the download of interest to the user.

- An ancillary protocol, called the PEX (for "Peer Exchange") protocol, was introduced to reduce the workload on the trackers. The PEX protocol allows a peer A to query peer B directly about the peers that B knows about that are currently in the swarm of interest to A. The PEX protocol opened up the possibility that P2P file sharing could go on even if the tracker were to go down on account of, say, a DoS attack.

- Of course, another shortcoming of the "traditional" approach is

that maintaining an index for all the `.torrent` files and the trackers can make the provider of these services potentially complicit in the violation of anti-piracy laws should the authorities discover these services as having facilitated unauthorized download of media content.

• So it should come as no surprise that torrent sites like `http://thepiratebay.org` have completely switched over to the DHT based operation of BitTorrent. With distributed storage and access made possible by DHT in the manner explained in the previous sections, there is now no need for centralized trackers anywhere. [With DHT, a BitTorrent client either directly downloads the hash of a torrent file or computes the same and then uses this hash as the content key to query the DHT for the node that has the tracker for that key. Subsequently, the client subscribes its IP address and the port number to that tracker. The tracker supplies to the client a list of all the peers currently in the swarm. The rest of the process is the same as with centralized trackers.]

• Even the need to provide a central index for the `.torrent` files is being done away with through the use of what are known as magnet links. A magnet link, at its simplest, is Base32 encoding of the SHA-1 hash of a `.torrent` file. Now instead of storing `.torrent` files directly and making them available through an index, a site such as `http://thepiratebay.org` would only store the magnet links and the BitTorrent clients would use those

links to search the DHT network for the node that has the tracker.

• Abandoning centralized trackers (and even abandoning central-
ized indexes for the torrent files) may make it easier for BitTorrent
service providers to stay one step ahead of the anti-piracy police,
the folks who like to use BitTorrent for downloading media con-
tent need to keep in mind the fact that nothing has changed from
the perspective of anyone being able to join a swarm and seeing
the IP addresses of all the others currently in the same swarm.

• Note that Ubuntu comes prepackaged with a BitTorrent client
that you are likely to find at Applications→Internet→BitTorrent.
Another popular BitTorrent client for Linux and Windows plat-
forms that we will mention later is BitTornado. Folks who use
MACS are likely to use a client called Miro.

• BitTorrent uses a set of policies to ensure a fast and fair distri-
bution of all the file pieces to all the peers in a swarm. Here are
some examples of these policies:

  – – Clients in a swarm request pieces for download in a random
    order to increase opportunities for trading pieces with other
    clients later.

–– It may seem that fair trading would result from a client sending pieces to only those clients who send pieces back. But such a policy, if followed strictly, would prevent new clients from joining a swarm. To get around this problem, a BitTorrent client uses what Bram Cohen has called **opportunistic unchoking**. This policy consists of a client using a portion of its bandwidth to send pieces to clients selected at random from the list made available by the tracker. This allows new BitTorrent clients to bootstrap themselves with information that they can subsequently trade.

• Each BitTorrent client keeps track of the other clients in a swarm. The set of the other clients known to a client is known as the **peer set**.

• BitTornado is a Python-based BitTorrent client. This client is also known as ShadowBT. This works on a one GUI per torrent basis. BitTornado is a set of command line utilities for working with BitTorrent files.

• To use BitTornado in the form of a GUI as a BitTorrent client, after downloading and installing the BitTornado package, all you have to do is to call

```
btdownloadgui  filename.torrent
```

or, if you want to specify the filename with just a command line
and without recourse to the GUI, use

```
btdownloadcurses  filename.torrent
```

assuming in both cases that you are invoking these commands in
a directory that contains the torrent file. To download a torrent
in the background, you can invoke

```
btdownloadheadless
```

• If you are using version 0.3.18 of BitTornado with wxPython for
the GUI, you may wish to look at the following "fix" provided by
me:

<div align="center">

`https://engineering.purdue.edu/kak/distbt/`

</div>

What you will find there is a rebundled BitTornado package with
changes to five files in order to make BitTornado 0.3.18 compat-
ible with the python-wxgtk2.8 package.

25.11: SECURITY ASPECTS OF STRUCTURED DHT-BASED P2P PROTOCOLS

- The basic protocols for open DHT-based overlay networks are founded on the assumption that every node joining the overlay can be trusted to provide its own $nodeID$ that can be assumed to come from a uniform probability distribution over the entire node identity space. For Chord and Pastry protocols, this space is the one-dimensional space corresponding to the Identifier Circle shown in Figure 1. When m bits are used for $nodeID$, this space will have a total of 2^m points in it. **In the rest of this section, we will use the phrase "identifier space" to refer to the space of all possible values for** $nodeID$ **and the content keys.**

- The above-stated founding assumption will in general be true if each node wishing to join a P2P network uses an algorithm such as SHA-1 or MD5 to hash its IP address into a fixed-length $nodeID$.

• In small P2P overlays, this trust in the participating nodes may be well-placed. But it would obviously be naive to make this assumption of trust if all and sundry are allowed to join a P2P overlay.

• When no constraints are placed on who can join a P2P overlay, security problems can be created by any or all of the following possibilities:

  – – a new node supplying a legitimate $nodeID$ but falsifying information in its own routing table

  – – a new node supplying a fake $nodeID$ that is meant to cause harm to the operation of the overlay

  – – the same new node joining an overlay repeatedly with different $nodeID$s

  – – a set of nodes conspiring together with fake values for $nodeID$ to disrupt the operation of the overlay

  We will now talk about each of the above possibilities.

• One of the easiest ways for a malicious node to cause problems is by falsifying the information in its routing table (and, for the

case of Pastry, in its leaf table also). As the reader will recall,
for the case of Chord, the ith entry in the routing table of the
node whose $nodeID$ is n is the IP address of the node whose
$nodeID$ is the smallest integer going clockwise after the point
$n + 2^{i-1}$ on the Identifier Circle. By inserting some other or even
a non-existent IP address at this location in the routing table,
routing queries would be misdirected (or not further directed at
all). This could cause the data to be stored at places from where
it would subsequently not be retrievable.

• All DHT-based overlays are vulnerable to false information in the
routing tables of the intruder nodes. This may be referred to as
a **topology attack** on a P2P overlay.

• Theoretically at least, the misdirections caused by fake pointers
in routing tables should be detectable because a query in a DHT-
based P2P overlay can only travel in the direction of decreasing
difference between the $nodeID$s and the content key. But for
this to actually work in practice, the propagation of a query in
an overlay must create an audit trail for the originator of the
query.

• Even with an audit trail, it may not be possible for the originator
of a query to verify that the query landed at the node whose

$nodeID$ is closest to the query key. That is because, by design,
DHT-based P2P overlays are meant to be a dynamic that allow
for nodes to join and leave at will and because, again by design,
there is no global record of the configuration of the overlay at
any time instant. So the only way to verify that data meant for
storage landed at the correct node is to later retrieve that data
from some other node in the overlay.

• Let's now consider the case when a node supplies a fake $nodeID$
when issuing its request to join the overlay. We will assume that
the intruder node is using a legitimate IP address assigned to it.
[Although not a part of the basic P2P protocols, a node's IP address could be verified by having it
acknowledge test messages when it first links up with its neighboring nodes in the P2P network. A
node advertising a fake IP address for itself could still receive test messages from other nodes in the
network if the fake address belonged to a co-conspirator machine. But, at least for the present, we will
assume that such is not the case.] An attack mounted with a fake $nodeID$,
especially if that identity belongs to some other legitimate node,
is called the **Spartacus Attack**.

• Let's further assume that the fake $nodeID$ supplied above is the
content key for a particular resource (that we may assume can
be computed by hashing either its title or its content). In this
manner, the node would become the destination for that content
object. If content keys are computed solely on the basis of a set
of key words or the title of the data object, the malicious node

could supply any questionable material when receiving a query
for that data object.

- Ordinarily, for the sake of fault tolerance and for dealing with
  node departures, replicas of the same data object would be stored
  at a set of nodes in a neighborhood (in the $nodeID$ space) of the
  node that minimizes the difference between the $nodeID$ and the
  key. Several nodes conspiring together could hijack a neighbor-
  hood around a key and cause disruptions with the delivery of any
  of the replicas. The same sort of an attack could be mounted by
  a single node that is able to field multiple $nodeID$ values. When
  a single malicious node presents multiple $nodeID$s to an overlay
  network, we say the offending node is mounting a **Sybil attack**.

- Another possible security problem can arise if a malicious node
  in a "legal" overlay is simultaneously a member of another sim-
  ilar overlay consisting of a set of co-conspiring malicious nodes.
  An unsuspecting new node wishing to join the legal overlay may
  instead get directed into the illegal overlay. If the illegal overlay
  contains some of the same data as the legal overlay, the new node
  may not be able to detect that anything is awry. Since the data
  storage in a P2P overlay is itself a dynamic process, in the sense
  that the data can migrate around as new nodes join and existing
  nodes leave the overlay, data siphoning off by illegal operators
  would not be detectable.

• Another security problem can occur when several malicious nodes decide to join and leave an overlay in rapid succession. This has the potential of degrading the performance of the overlay network since the routing table updates at all the affected nodes in the overlay may not be able to keep up with the additions and the departures of the offending nodes. As a result, several legitimate nodes may end up with inconsistent routing tables.

• Proximity routing used in Pastry is vulnerable to fake proximities injected into the overlay by a malicious node working in cahoots with other malicious nodes. Ordinarily, an estimate of proximity would be obtained by invoking a utility such as `traceroute` to estimate the number of a hops to a given IP. But if this probe is intercepted by a malicious node, that node can send back a pointer to another cooperating malicious node.

• Although not by itself a security issue, the fact that it is now common for a machine to possess a non-static IP addresses (through DHCP) can create issues of its own with regard to how a node behaves in an overlay. Let's say an active node changes its IP address after its DHCP lease expires, that would invalidate its IP-address-based $nodeID$. Suddenly, all of the information stored at the node would become inconsistent with its new $nodeID$. So the node would have to reinitialize itself as if it was starting with a blank slate.

• Somewhat along the same lines as mentioned above, the fact
  that many machines these days operate behind NAT devices and
  proxy servers can also create big problems with regard to their
  participation in DHTs. The IP addresses for these machines as
  visible from the outside are usually the same for all the machines
  that are being NATed or that are operating behind the same
  proxy server. So it may make no sense to base the *nodeID* for
  such machines on their IP addresses.

• To deal with the above problem and also to make it easier to
  authenticate the nodes participating in a P2P overlay network,
  it has been suggested that the *nodeID* of a node be derived by
  hashing its public key. **Such a** *nodeID* **would be unfor-
  gable**.

• About defenses against the security problems mentioned above
  (and others not mentioned here), P2P security is still a wide open
  research area. As P2P system become even more important in
  the years to come, the security aspect will surely see a lot of
  action.

• It is conceivable that as a protection against some of the attacks
  listed above, a structured P2P network will have certain desig-
  nated nodes acting as **guards at certain chosen locations**

in the identifier space. By exchanging "network integrity messages" amongst themselves — messages that involve different values for the content keys — and observing the behavior of the overlay network with regard to the storage and retrieval of those messages, the guard nodes will be able to monitor the health of the overlay.

- Further protection could be obtained by designating certain trusted machines to act as bootstrap machines. Those would be the only entry points for the new nodes. In order not to create choke points in a P2P system, the set of machines designated for boot-strapping could itself be made dynamic by insisting that such machines possess certificates issued by certain authorities.

25.12: ANONYMITY IN STRUCTURED P2P OVERLAY NETWORKS

• There is a legitimate need for privacy and anonymity in the conduct of human enterprise. That is, perverts and the mentally sick are not the only ones who may wish to remain private and/or anonymous in their dealings with the rest of the world.

• Privacy and anonymity are somewhat interrelated. Whereas privacy refers to a desire that others not become privy to one's actions, anonymity refers to one engaging in actions that would be visible to others but without a knowledge of the author of the actions.

• As an example of a legitimate need for privacy, it is now common for lawyers to ask their expert witnesses to not engage in any meaningful email communication with the lawyers because all email can be discovered and subject to court scrutiny. So if an expert witness wants to engage the law firm he/she is working for in any deep dialog about the issues, it can only be done either through voice communications or in face-to-face meetings. If it

was possible for there to be a form of email that would allow
people to communicate as privately as,say, through a telephone
call (no wiretapping assumed), it would be popular in many le-
gitimate business enterprises.

- With regard to anonymity, history has shown that it serves an im-
portant role in legitimate expressions of dissent and in mounting
opposition to repressive control.

- Privacy and anonymity are also important for each one of us
individually in order to keep our private lives private — especially
with regard to how we interact with the world of the internet. It
should be no one's business as to what sort of music and movies
I download from web, or as to which web sites I frequent for my
amusement. [Individually we desire privacy and anonymity, not because we have
anything illegitimate to hide, but simply because we do not care for others to know
about certain aspects of ourselves. For example, I have a desire to not be seen by others
when I am picking my nose.]

- Anonymity was one of the original reasons for people to get ex-
cited about P2P networks. The reasoning was that if information
could migrate to any node (or to even a set of nodes) in a P2P
network and could subsequently be downloaded from wherever it
resided, there would be less of an association between the infor-

mation and the owner of that information, and therefore less of
a legal hassle associated with the download of that information.

• So an important question is as to what extent the structured P2P
overlay networks provide anonymity to the nodes participating in
a network.

• A problem with most structured P2P protocols is that in their
basic implementations they do not allow for the nodes to remain
anonymous. Since the overlay topology is controlled strictly by
mathematical formulas and since that is also the case with how
the information to be stored is assigned to the different nodes,
ordinarily speaking there is no anonymity either with regard to
the node identities or with regard to the association between the
nodes and the information they make available.

• Yet, it is possible for a DHT-based structured P2P overlay to
provide **sender anonymity** — as demonstrated recently by
the work of Nikita Borisov — provided the lookup queries are
forwarded **recursively** as opposed to **iteratively**.

• We say that a lookup query in a P2P overlay **is forwarded
iteratively** if the original sender node contacts one of its neigh-

bors in its routing table to find out where the query should be directed. Subsequently, the original node sends the query to that node to find out where the query should be sent next. This process continues iterative until the goal node is reached that minimizes the distance between the key and the *nodeID*. At that point, the original sender sends the query directly to the goal node. Note that, in the iterative mode, the original sender node directly communicates with all of the enroute nodes until the goal node is found. **Obviously, there is no way for the sender to remain anonymous in this approach to data lookup for either posting new information or retrieving existing information.**

- We say that a lookup query in a P2P overlay **is forwarded recursively** if the original sender node sends the query itself to one of its neighbors in its routing table. That node forwards the query to the next node in its own routing table, and so on, until the query reaches the goal node. If the query was for posting new information, there would be no need to include the identity of the original sender with the query as it wends its way to the final destination. On the other hand, if the query was for retrieving data, the destination node (if it has the data) can send the data back to the node from which it received the query, and that node can send it back to where it got the query from, and so on. In this manner, the original sender of the query will be able to access the information but the intermediate nodes will not have direct access to the identity of the original sender. **So this**

**approach does offer a measure of sender anonymity
as the intermediate nodes would not know where the
query originated.**

- But the above seemingly simple approach to achieving anonymity
has a few shortcomings, not the least of which is that it depends
on the distance between the original sender and the final desti-
nation in the identifier space.

- Borisov has shown that it is possible to achieve greater anonymity
with recursive queries if one interposes a random walk in the path
of a query from the original sender to its final destination.

25.13: AN ANSWER TO "Will I be Caught?"

• When P2P file-sharing tools first hit the internet, they became instantly popular. Many people believed that a big reason for this popularity was the perception that there was less of a chance of "getting caught" if you downloaded music or video from others just like yourself, especially if the different pieces of what you wanted came from different randomly selected places on the internet.

• So are the chances of being caught really less with a P2P file-sharing system compared to the more traditional methods of downloading stuff directly from web sites?

• The bottom-line answer to the question is that you are just as vulnerable with P2P as you are with the more traditional methods — **even if what you are downloading is protected by strong encryption**.

● Confidentiality offered by strong encryption and anonymity of-
fered by mechanisms such as the insertion of random walks in
query propagation in P2P overlays is mostly illusory.

● If you can join a P2P network, so can any everyone else and
that includes the folks who want to know what it is you are
downloading. If your viewer is able to see the contents of an
encrypted file, the viewer available to those folks will be able to do
the same. If the material is questionable, all that the enforcement
folks have to do is to compare the digital signature of the file they
downloaded with the file you downloaded. So giving a different
name to a downloaded encrypted file with questionable content
does not necessarily protect you.

● You could, of course, give the files strong encryption on your own
with a key that only you know about. Obviously, such a file would
not be examinable by enforcement folks. But, don't forget that
when you downloaded the file it was encrypted either by the P2P
protocol or by a protocol associated with the communication link.
If the encryption was provided by the P2P protocol with a session
key (that may be different for each download), your end of the
protocol most likely decrypted the file before it was stored on the
disk. The same would be true for the encryption provided by the
communication link. In either case, the ISP you are connected to
can easily record the digital signatures of the files you download.

These can be compared with the digital signatures of the files
that the enforcement folks would download from the same P2P
network.

- I am not saying that ISPs routinely log the digital signatures
  of the files that are downloaded by their customers. But the
  important point is that it *could* be done.

- The enforcement folks may also deliberately post questionable
  material in the overlay database to catch "bad guys." Not only
  that, people who want to watch you and others can deploy a
  large number of machines at different points in a P2P overlay's
  identifier space to monitor how the information is being routed
  in those portions of the space. By examining the routing tables
  at the nodes under their control, the enforcement folks can get
  a sense of how the material posted by them is migrating in the
  overlay. This may not immediately reveal as to who has actually
  downloaded what. However, since the routing tables must contain
  the IP addresses, over time and with repeated trials of the ploy,
  the enforcement folks may be able develop a good list of suspects.

- The bottom line is that there is practically no anonymity on the
  internet. [Even using fake login names at popular web email sites does not give you a whole
  lot of anonymity since the locations from where you are logging in can be monitored.]

• Yes, by using strong encryption, you can get a measure of confidentiality, but that is only true amongst a small group of individuals who trust one another. There is no privacy or confidentiality when it comes to downloading files that are available to all and sundry even when these files are strongly encrypted.

• If what I have said above is the case, how come there are so many bad guys on the internet? From all the spam we receive and all of the nefarious places we can wander into, there appears to be no shortage of scamsters on the internet. How can they get away with it?

• The answer to the above question is that law enforcement with regard to internet fraud is extremely weak in a large number of countries. Additionally, even in the US, the law enforcement folks simply have not considered it worth their while to pursue certain crimes on the internet.

• So if you have a desire to use P2P for questionable downloads, I'd say don't. It's not worth it.

25.14: SUGGESTIONS FOR FURTHER READING

- Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications" SIGCOMM'01, August 27-31, San Diego, CA, 2001.

- Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, "Looking Up Data in P2P Systems," Communications of ACM, pp. 43-47, 2001.

- Antony Rowstron and Peter Druschel, "Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems," 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Heidelberg, Germany, 2001.

- Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron, "Exploiting Network Proximity in Peer-to-Peer Overlay Networks," Technical Report MSR-TR-2002-82, Microsoft Research, 2002.

- Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron, "Exploiting Network Proximity in Distributed Hash Tables," position paper.

- Emil Sit and Robert Morris, "Security Considerations for Peer-to-Peer Distributed Hash Tables," Proc. of the 1st International Workshop on Peer-to-Peer Systems (IFTPS '02), Cambridge, MA, March 2002.

- John Douceur, "The Sybil Attack," Proc. of the 1st International Workshop on Peer-to-Peer Systems (IFTPS '02), Cambridge, MA, March 2002.

- Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan Wallach, "Secure Routing for Structured Peer-to-Peer Overlay Networks," Proc. 5th Usenix Symposium on Operating Systems Design and Implementation, December 2002.

- Nikita Borisov, "Anonymous Routing in Structured Peer-to-Peer Overlays", Ph.D. Dissertation, Computer Science, University of California, Berkeley, 2005.

- Peter Maymounkov and David Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," Proceedings of IPTPS02, Cambridge, March 2002.

- Michael Goodrich, Michael Nelson, and Jonathan Sun, "The Rainbow Skip Graph: A Fault-Tolerant Constant-Degree Distributed Data Structure," Proceedings of the 7th Annual ACM/SIAM Symposium on Discrete Algorithms, pp. 384-393, New York, 2006.

- Nathan Evans, Chris GauthierDickey, and Christian Grothoff, "Routing in the Dark: Pitch Black," ACSAC 2007.