CHORD

• Chord is a protocol implementing for a peer-to-peer distributed hash table
• A distributed hash table stores key-value pairs by assigning keys to different computers (nodes)
• A node stores the values for all the keys it is responsible for
• Chord specifies how keys are to be assigned to nodes, and how a node can discover the value for a given key by first locating the node responsible for that key
• Chord is one of the four original distributed hash table protocols, along with CAN, Tapestry, and Pastry

The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
CHORD

• Peer-to-peer systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality

• The core operation in most peer-to-peer systems is efficient location of data items

• Chord is a scalable protocol for lookup in a dynamic peer-to-peer system with frequent node arrivals and departures
CHORD

• The Chord protocol supports just one operation: given a key, it maps the key onto a node

• Depending on the application using Chord, that node might be responsible for storing a value (address, a document, or an arbitrary data item) associated with the key

• Chord uses a variant of consistent hashing to assign keys to Chord nodes
CHORD

- **Consistent hashing**
  - tends to balance load, since each node receives roughly the same number of keys, and
  - involves relatively little movement of keys when nodes join and leave the system
- Each Chord node needs “routing” information about only a few other nodes
- Because the routing table is distributed, a node resolves the hash function by communicating with a few other nodes
- In the steady state, in an $N$ node system, each node maintains information only about $O(\log(N))$ other nodes and resolves all lookups via $O(\log(N))$ messages to other nodes
- Chord maintains its routing information as nodes join and leave the system
- With high probability each such event results in no more than $O(\log(N)^2)$ messages

The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
CHORD

• Chord maps keys onto nodes, traditional name and location services (e.g. DNS) provide a direct mapping between keys and values

• A value can be an address, a document, or an arbitrary data item
CHORD

- DNS provides a host name to IP address mapping
- Chord requires no special servers, while DNS relies on a set of special root servers
- DNS names are structured to reflect administrative boundaries
- Chord imposes no naming structure
- DNS is specialized to the task of finding named hosts or services
- Chord can also be used to find data objects that are not tied to particular machines

The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
CHORD

- The Chord software takes the form of a library to be linked with the client and server applications that use it.
- The application interacts with Chord in two main ways:
  - First, Chord provides a `lookup(key)` algorithm that yields the IP address of the node responsible for the key.
  - Second, the Chord software on each node notifies the application of changes in the set of keys that the node is responsible for.
CHORD

• At its heart, Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them

• It uses **consistent hashing**, which has several good properties
  - With high probability the hash function balances load (all nodes receive roughly the same number of keys)
  - With high probability, when an $N$th node joins (or leaves) the network, only an $O(1/N)$ fraction of the keys are moved to a different location (this is clearly the minimum necessary to maintain a balanced load)
CHORD

• Chord improves the scalability of consistent hashing by avoiding the requirement that each node knows about every other node
• A Chord node needs only a small amount of “routing” information about other nodes
• Because this information is distributed, a node resolves the hash function by communicating with a few other nodes
• In an $N$-node network, each node maintains information only about $\log(N)$ other nodes, and a lookup requires $O(\log(N))$ messages
• Chord must update the routing information when a node joins or leaves the network
• Join or leave requires $O(\log(N)^2)$ messages

The material of this section is mainly drawn from: Sigcomm '01: “Chord: A Scalable Peertopeer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan

**CHORD-consistent hashing**

- The consistent hash function assigns each node and key an $m$-bit identifier using a base hash function such as SHA-1
- A node’s identifier is chosen by hashing the node’s IP address, while a key identifier is produced by hashing the key
- We will use the term “key” to refer to both the original key and its image under the hash function
- Similarly, the term “node” will refer to both the node and its identifier under the hash function
- The identifier length must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible
CHORD-consistent hashing

- Consistent hashing assigns keys to nodes as follows
- Identifiers are ordered in an identifier circle modulo $2^m$
- Key $k$ is assigned to the first node whose identifier is equal to or follows the identifier of $k$ in the identifier space
- This node is called the successor node of key $k$, denoted by $\text{successor}(k)$
- If identifiers are represented as a circle of numbers from 0 to $2^{m-1}$, then $\text{successor}(k)$ is the first node clockwise from $k$

The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
CHORD-consistent hashing

• The figure shows an identifier circle with $m = 3$
• The circle has three nodes: 0, 1, and 3
• The successor of identifier 1 is node 1, so key 1 would be located at node 1
• Similarly, key 2 would be located at node 3, and key 6 at node 0
CHORD-consistent hashing

- Consistent hashing is designed to let nodes enter and leave the network with minimal disruption.
- To maintain the consistent hashing mapping when a node $n$ joins the network, certain keys previously assigned to $n$'s successor now become assigned to $n$.
- When node $n$ leaves the network, all of its assigned keys are reassigned to $n$'s successor.
- No other changes in assignment of keys to nodes need occur.
- In the example, if a node were to join with identifier 7, it would capture the key with identifier 6 from the node with identifier 0.

The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan.
CHORD-scalable key location

- Let $m$ be the number of bits in the key/node identifiers
- Each node $n$ maintains a routing table with (at most) $m$ entries, called the finger table
- The $k$th entry in the table at node $n$ contains the identity of the first node $s$ that succeeds $n$ by at least $2^{k-1}$ on the identifier circle, i.e., $s = \text{successor}(n + 2^{k-1})$
- We call node $s$ the $k$th finger of node $n$, and denote it by $n.\text{finger}[k].\text{node}$
- A finger table entry includes both the Chord identifier and the IP address (and port number) of the relevant node
- Note that the first finger of $n$ is its immediate successor on the circle
- For convenience we often refer to it as the successor rather than the first finger

This is an entry of the node’s $n$ finger table
This table is referred to as $n.\text{finger}$
The $i$th entry of the table is $n.\text{finger}[i]$

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{finger}[k].\text{start}$</td>
<td>$(n + 2^{k-1}) \mod 2^m, 1 \leq k \leq m$</td>
</tr>
<tr>
<td>$\text{interval}$</td>
<td>$[\text{finger}[k].\text{start}, \text{finger}[k+1].\text{start}]$</td>
</tr>
<tr>
<td>$\text{node}$</td>
<td>first node $\geq n.\text{finger}[k].\text{start}$</td>
</tr>
<tr>
<td>$\text{successor}$</td>
<td>the next node on the identifier circle; $\text{finger}[1].\text{node}$</td>
</tr>
<tr>
<td>$\text{predecessor}$</td>
<td>the previous node on the identifier circle</td>
</tr>
</tbody>
</table>
CHORD-scalable key location

- In the example, the finger table of node 1 points to the successor nodes of identifiers:
  - \((1+2^0) \mod 8 = 2\)
  - \((1+2^1) \mod 8 = 3\)
  - \((1+2^2) \mod 8 = 5\)

- The successor of identifier 2 is node 3
- The successor of identifier 3 is node 3
- The successor of identifier 5 is node 0

The material of this section is mainly drawn from: Sigcomm '01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
CHORD-scalable key location

- This scheme has two important characteristics
- First, each node stores information about only a small number of other nodes, and knows more about nodes closely following it on the identifier circle than about nodes farther away
- Second, a node’s finger table generally does not contain enough information to determine the successor of an arbitrary key
- For example, node 3 does not know the successor of 1, as 1’s successor (node 1) does not appear in node 3’s finger table

The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan.
CHORD-scalable key location

- What happens when a node \( n \) does not know the successor of a key \( k \)?
- If \( n \) can find a node whose ID is closer than its own to \( k \), that node will know more about the identifier circle in the region of \( k \) than \( n \) does.
- Thus, \( n \) searches its finger table for the node \( j \) whose ID most immediately precedes \( k \), and asks \( j \) for the node it knows whose ID is closest to \( k \).
- By repeating this process, \( n \) learns about nodes with IDs closer and closer to \( k \).

The material of this section is mainly drawn from: Sigcomm '01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan.
CHORD-scalable key location

- As an example, consider the Chord ring in Figure
- Suppose node 3 wants to find the successor of identifier 1
- Since 1 belongs to the circular interval [7, 3), it belongs to 3.finger[3].interval
- Node 3 therefore checks the third entry in its finger table, which is 0
- Because 0 precedes 1, node 3 will ask node 0 to find the successor of 1
- In turn, node 0 will infer from its finger table that 1’s successor is the node 1 itself, and return node 1 to node 3
CHORD-scalable key location

- The finger pointers at repeatedly doubling distances around the circle cause each iteration of the loop to halve the distance to the target identifier.

- From this intuition follows a theorem:

  *With high probability, the number of nodes that must be contacted to find a successor in an n-node network is $O(\log(N))$*
CHORD-node joins

• In a dynamic network, nodes can join (and leave) at any time
• The main challenge in implementing these operations is preserving the ability to locate every key in the network
• To achieve this goal, Chord needs to preserve two invariants:
  – 1. Each node’s successor is correctly maintained
  – 2. For every key $k$, node $successor(k)$ is responsible for $k$
• In order for lookups to be fast, it is also desirable for the finger tables to be correct
• With high probability, any node joining or leaving an $N$-node Chord network will use $O(\log^2 N)$ messages to re-establish the Chord routing invariants and finger tables
CHORD-node joins

• To simplify the join and leave mechanisms, each node in Chord maintains a *predecessor pointer*

• A node’s predecessor pointer contains the Chord identifier and IP address of the immediate predecessor of that node

• It can be used to walk counterclockwise around the identifier circle
CHORD-node joins

• To preserve the invariants stated above, Chord must perform three tasks when a node $n$ joins the network:
  - 1. Initialize the predecessor and fingers of node $n$
  - 2. Update the fingers and predecessors of existing nodes to reflect the addition of $n$
  - 3. Notify the higher layer software so that it can transfer state (e.g. values) associated with keys that node $n$ is now responsible for
CHORD-node joins

• We assume that the new node learns the identity of an existing Chord node $n'$ by some external mechanism

• Node $n$ uses $n'$ to initialize its state and add itself to the existing Chord network, in three phases as follows
CHORD-node joins

- **Phase 2:** *Updating fingers of existing nodes*
- Node \( n \) will need to be entered into the finger tables of some existing nodes
- For example, in the Figure, node 6 joins and becomes the third finger of nodes 0 and 1, and the first and the second finger of node 3

The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
CHORD-stabilization

• If joining nodes have affected some region of the Chord ring, a lookup that occurs before stabilization has finished can exhibit one of three behaviors
• The common case is that all the finger table entries involved in the lookup are reasonably current, and the lookup finds the correct successor in $O(\log N)$ steps
• The second case is where successor pointers are correct, but fingers are inaccurate
• This yields correct lookups, but they may be slower
• In the final case, the nodes in the affected region have incorrect successor pointers, or keys may not yet have migrated to newly joined nodes, and the lookup may fail
• The higher-layer software using Chord will notice that the desired data was not found, and has the option of retrying the lookup after a pause
• This pause can be short, since stabilization fixes successor pointers quickly

The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan
Eclipse attacks on CHORD

• In an “Eclipse” attack, a set of malicious, colluding overlay nodes arranges for a correct node to peer only with members of the coalition
• If successful, the attacker can mediate most or all communication to and from the victim
• Furthermore, by supplying biased neighbor information during normal overlay maintenance, a modest number of malicious nodes can eclipse a large number of correct victim nodes

The material of this section is mainly drawn from: Sigcomm ’01: “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications”, by Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan