Distributed Hash Tables

What is a DHT?

- Hash Table
  - data structure that maps “keys” to “values”
  - essential building block in software systems
- Distributed Hash Table (DHT)
  - similar, but spread across many hosts
- Interface
  - insert(key, value)
  - lookup(key)

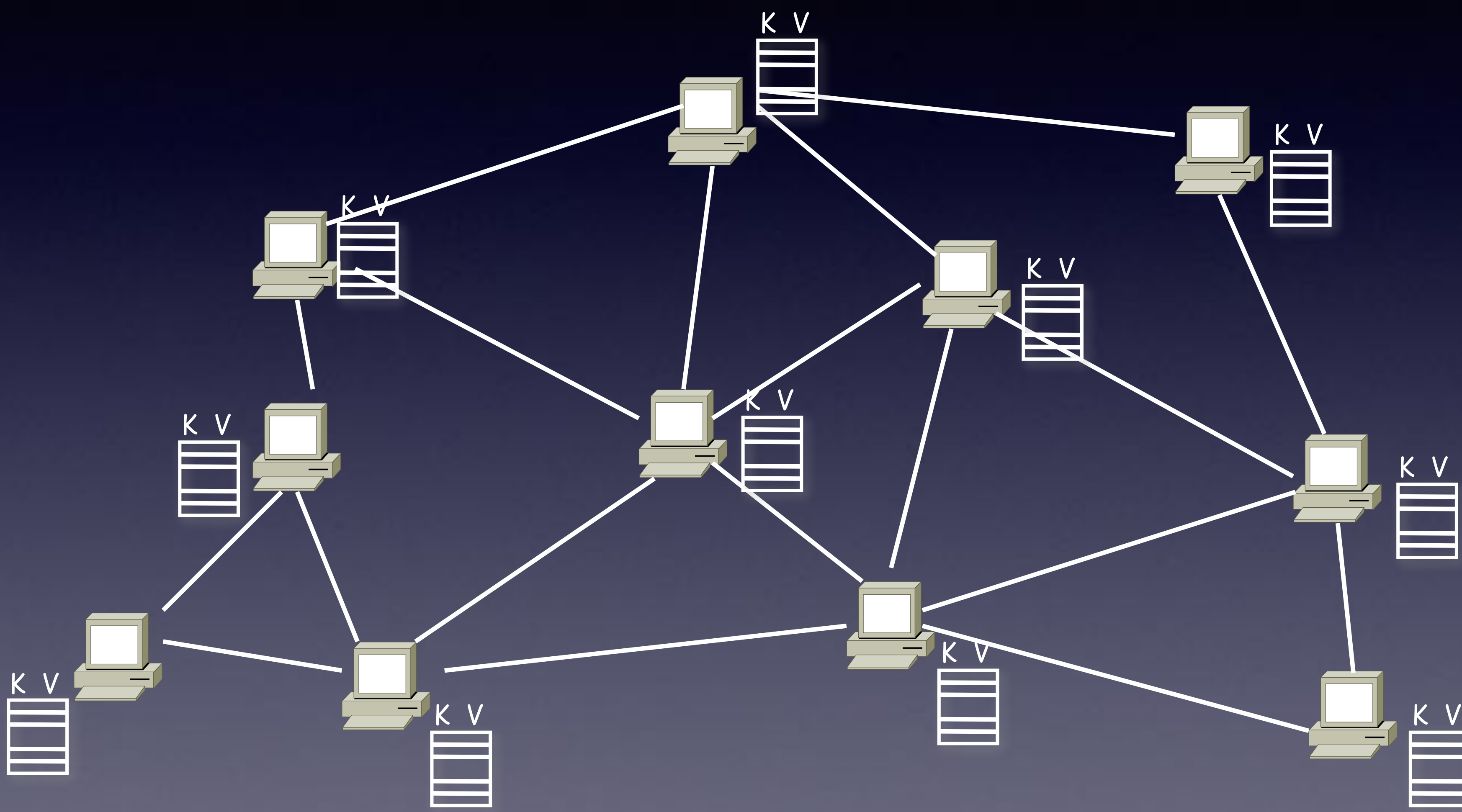
How do DHTs work?

Every DHT node supports a single operation:

- Given *key* as input; route messages to node holding *key*
- DHTs are *content-addressable*

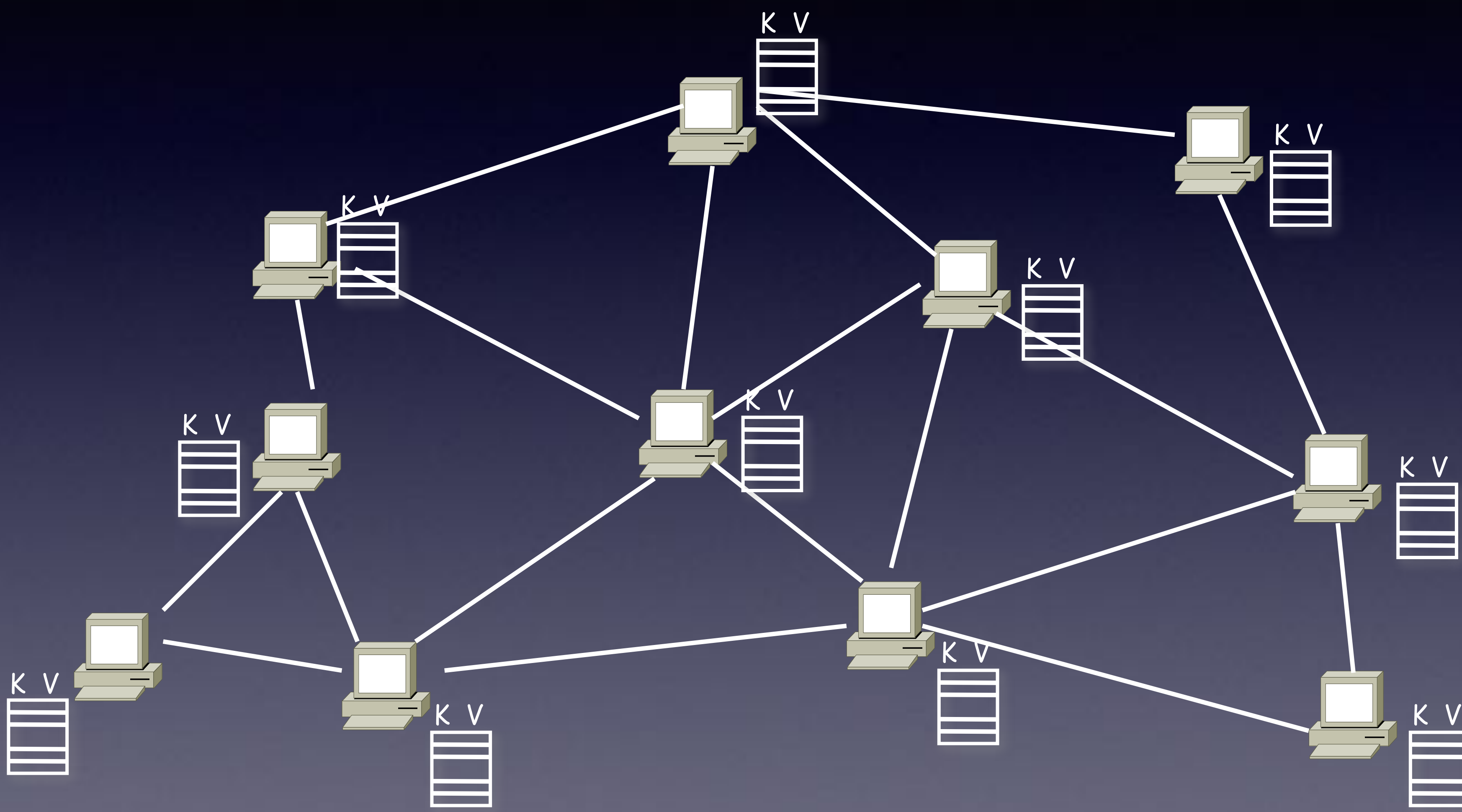
DHT: basic idea

DHT: basic idea



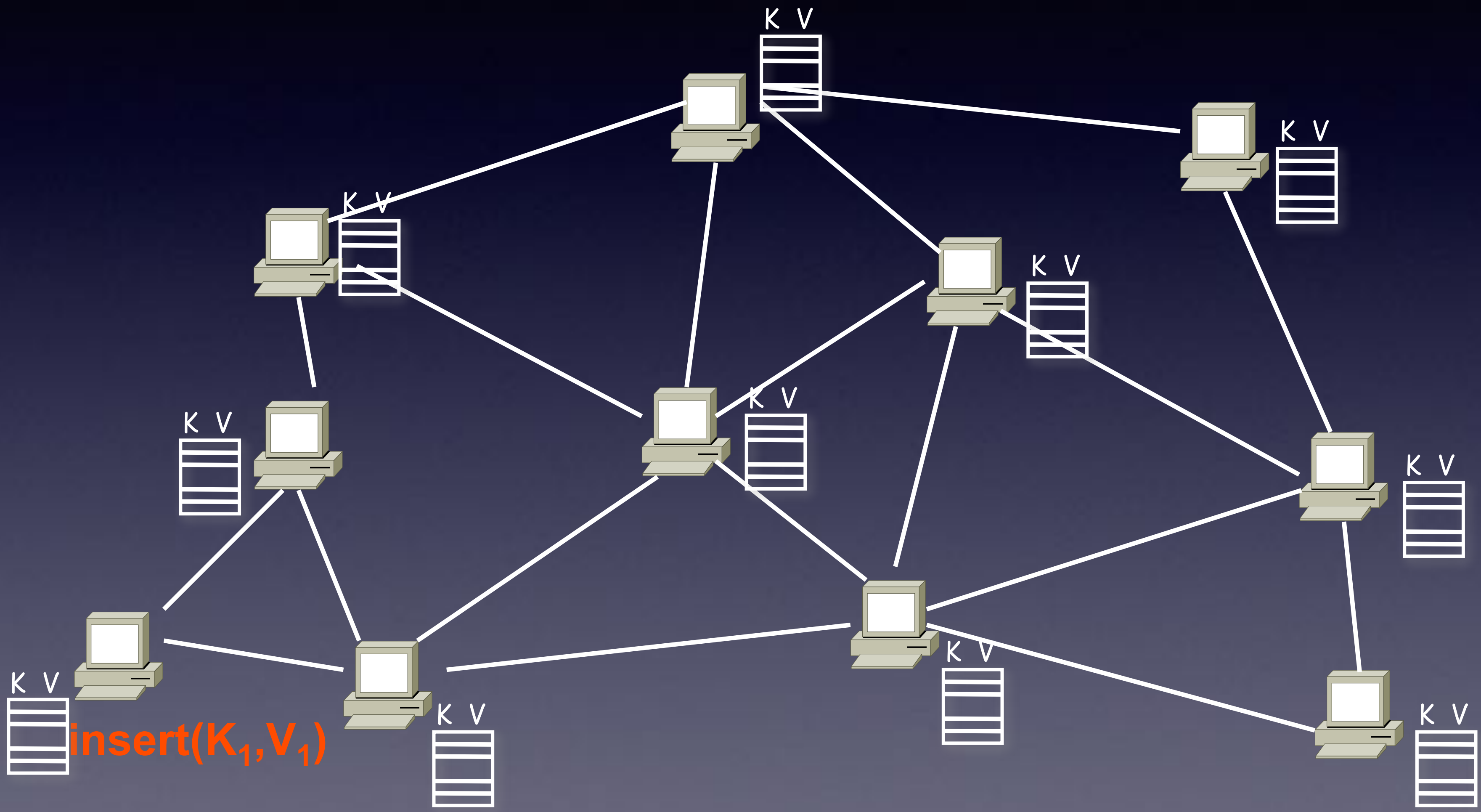
Neighboring nodes are "connected" at the application-level

DHT: basic idea



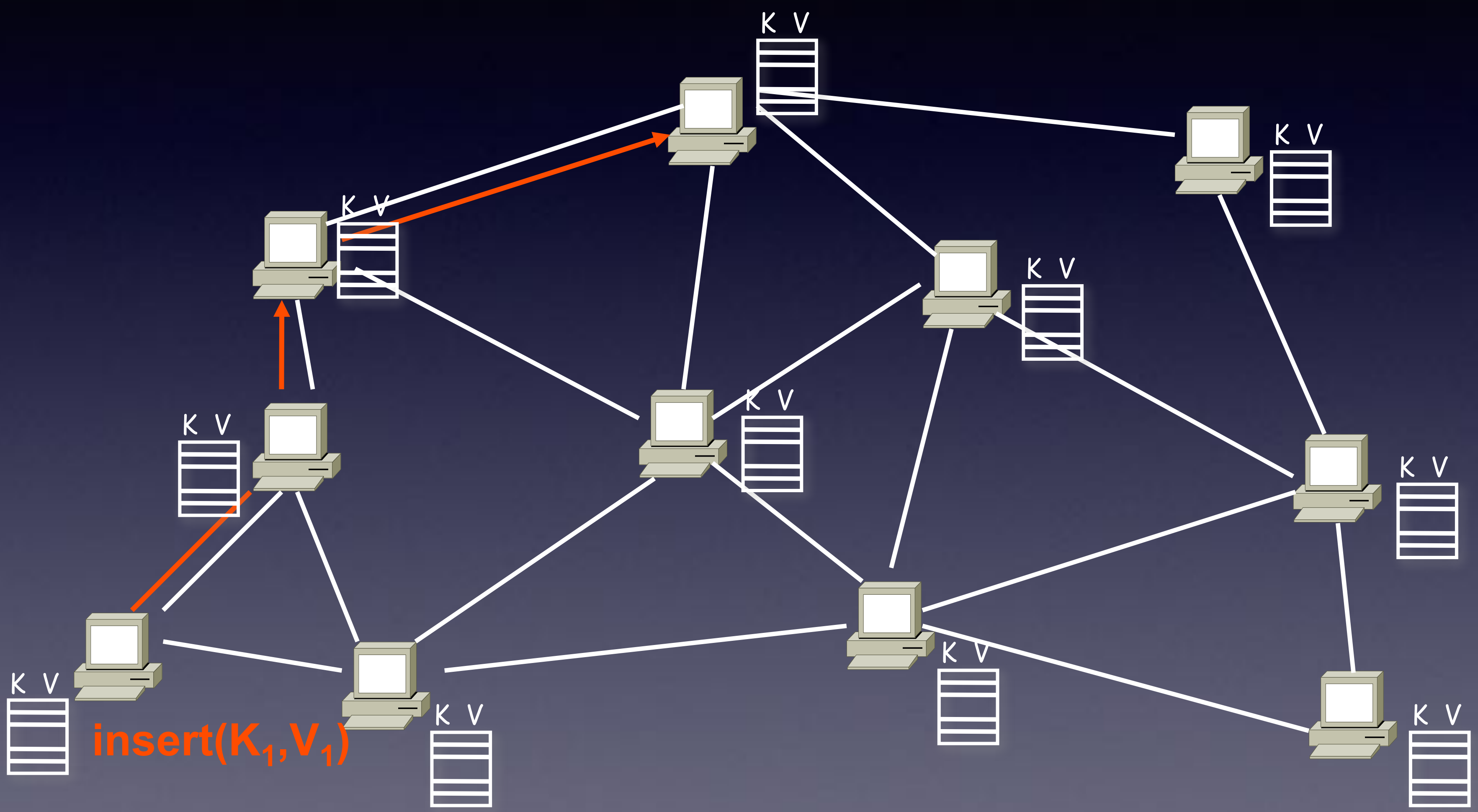
Operation: take *key* as input; route messages to node holding *key*

DHT: basic idea



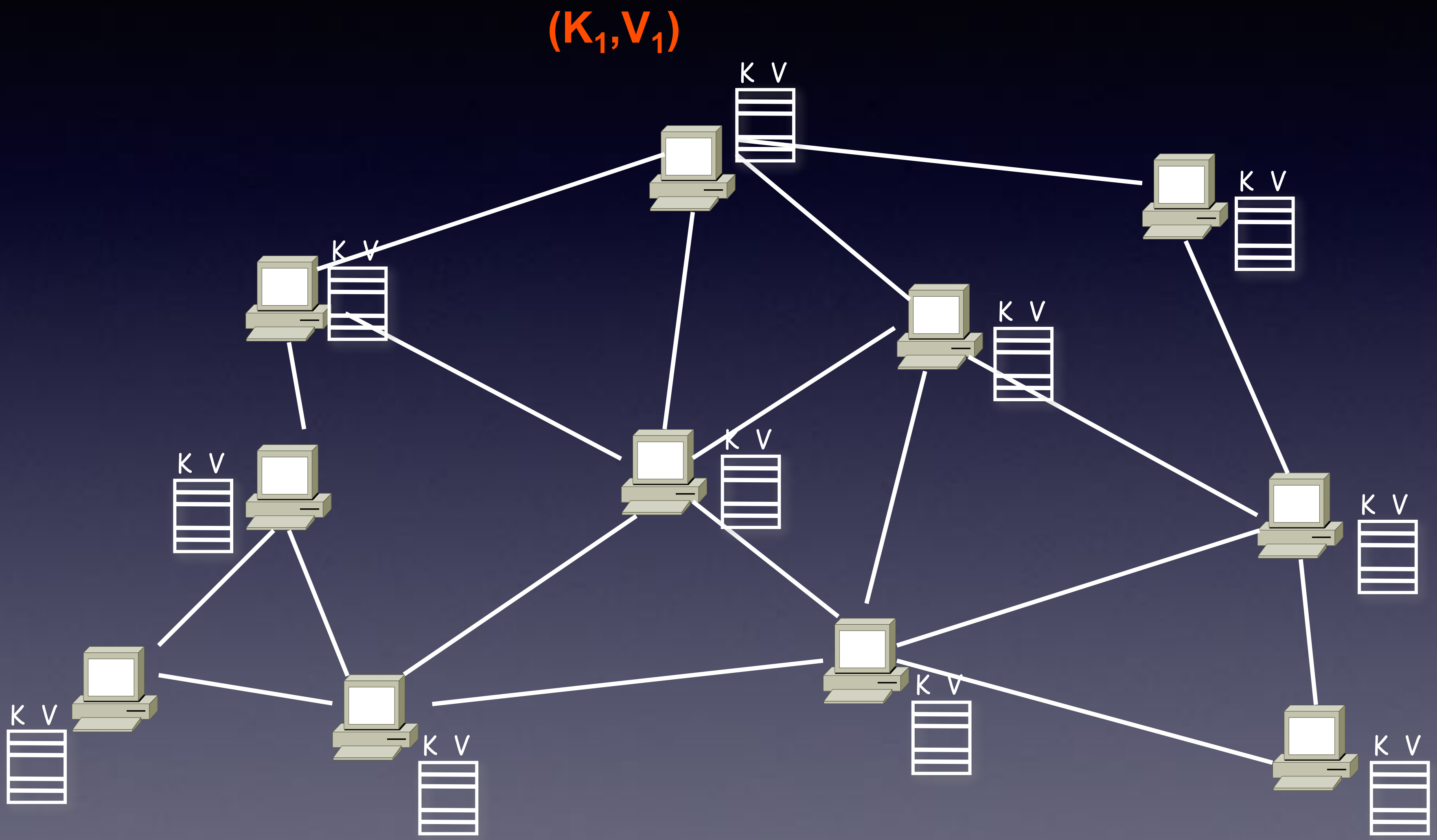
Operation: take *key* as input; route messages to node holding *key*

DHT: basic idea



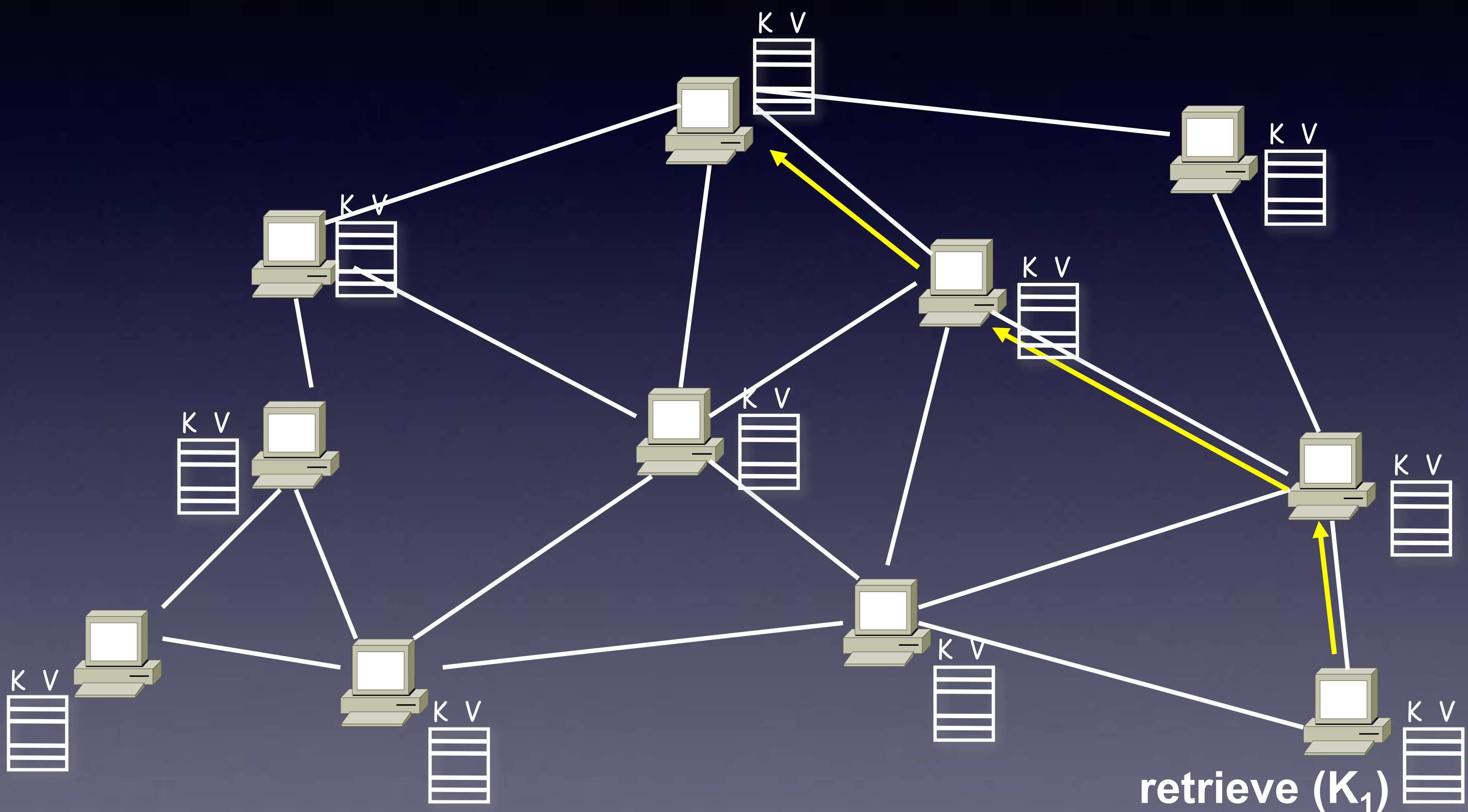
Operation: take *key* as input; route messages to node holding *key*

DHT: basic idea



Operation: take *key* as input; route messages to node holding *key*

DHT: basic idea

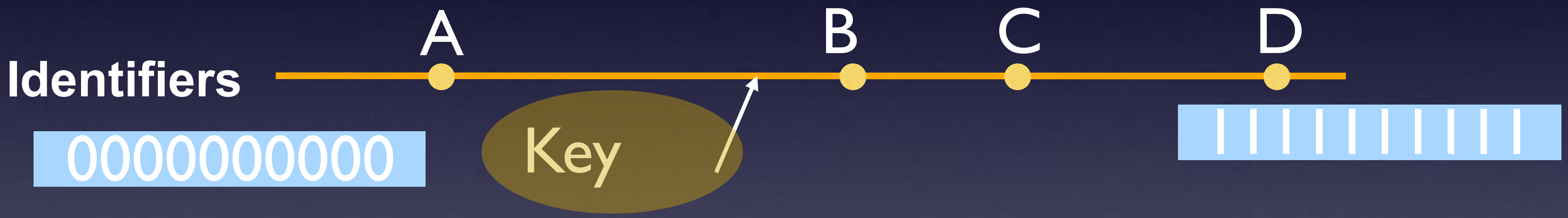


Operation: take *key* as input; route messages to node holding *key*

- For what settings do DHTs make sense?
  - Why would you want DHTs?

Fundamental Design Idea I

- Consistent Hashing
  - Map keys *and* nodes to an *identifier* space; implicit assignment of responsibility



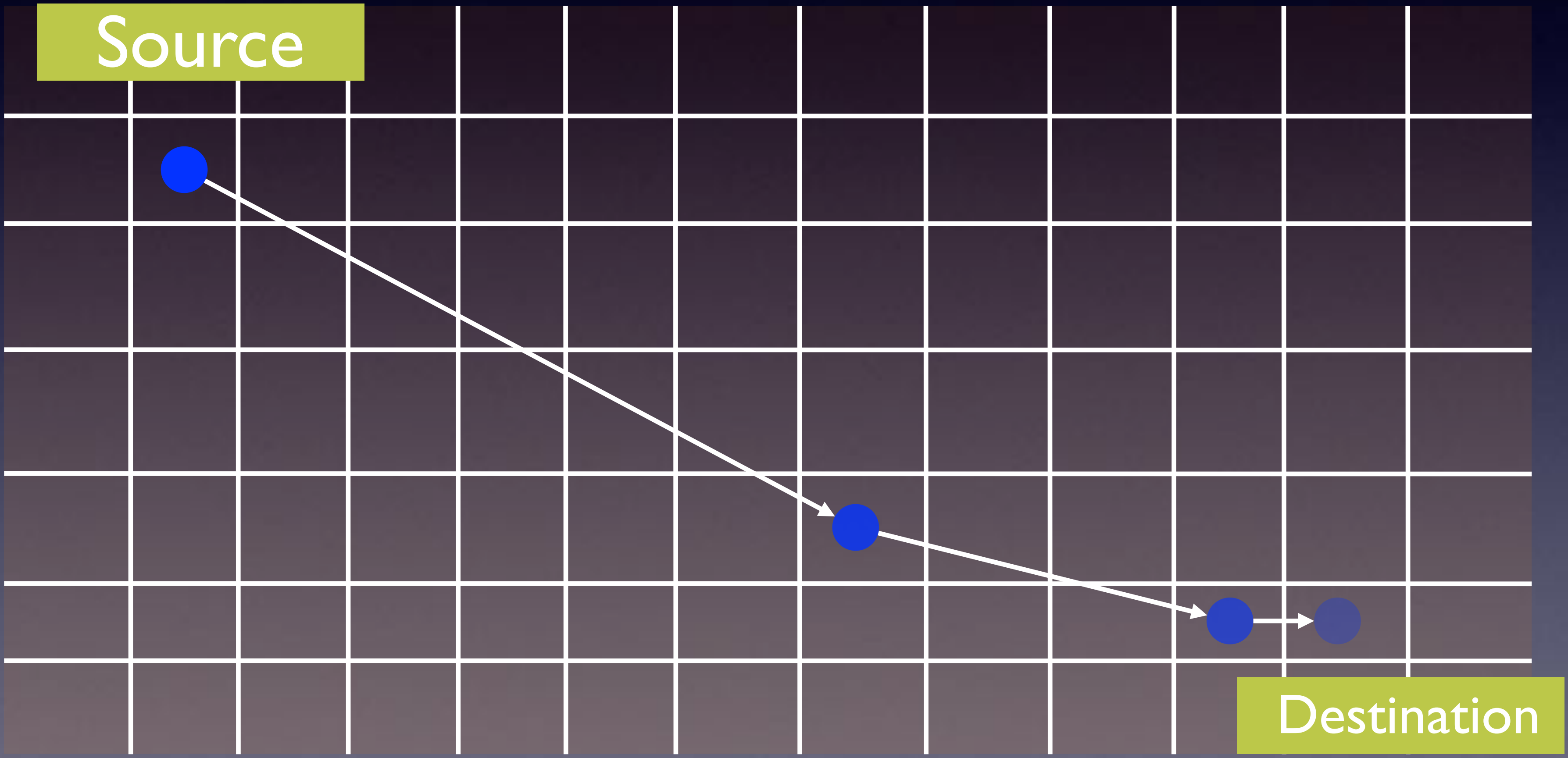
Mapping performed using hash functions (e.g., SHA-1)

- What is the advantage of consistent hashing?

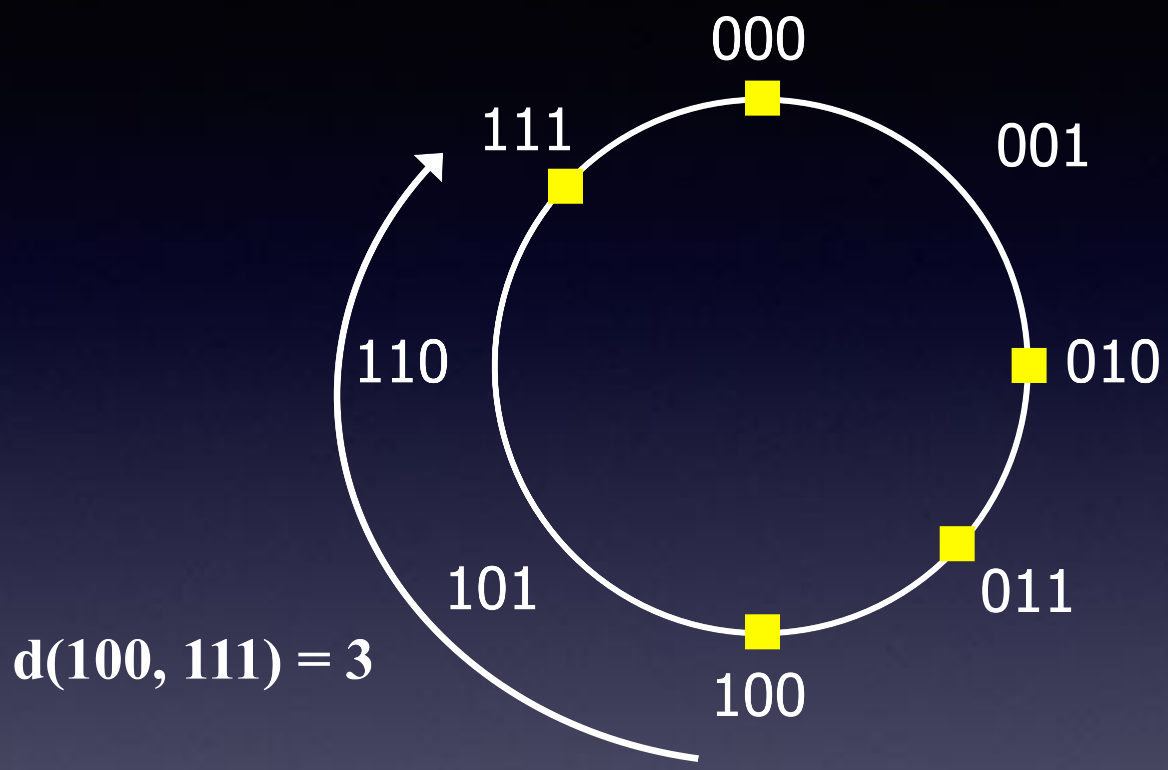
Consistent Hashing

Fundamental Design Idea II

- Prefix / Hypercube routing

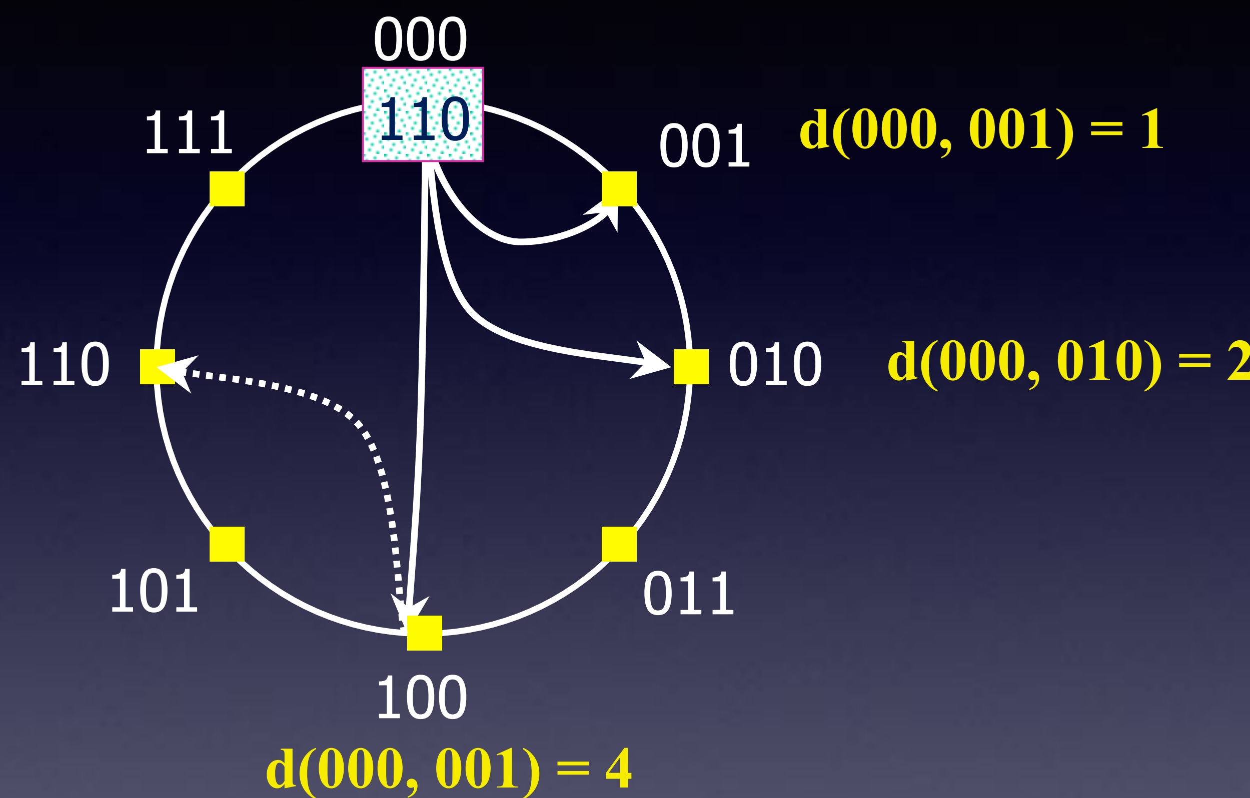


State Assignment in Chord



- Nodes are randomly chosen points on a clock-wise ring of *values*
- Each node stores the *id space* (*values*) between itself and its predecessor

Chord Topology and Route Selection



- Neighbor selection: **i**th neighbor at **2**^{i} distance
- Route selection: pick neighbor closest to destination

Joining Node

- Assume system starts out w/ correct routing tables.
- Use routing tables to help the new node find information.
  - New node m sends a lookup for its own key
  - This yields m.successor
  - m asks its successor for its entire finger table.
  - Tweaks its own finger table in background
  - By looking up each m + 2^i

Routing to new node

- Initially, lookups will go to where it would have gone before m joined
- m's predecessor needs to set successor to m. Steps:
  - Each node keeps track of its current predecessor
  - When m joins, tells its successor that its predecessor has changed.
  - Periodically ask your successor who its predecessor is:
  - If that node is closer to you, switch to that guy.
  - this is called "stabilization"
- Correct successors are sufficient for correct lookups!

Concurrent Joins

- Two new nodes with very close ids, might have same successor.
- Example:
  - Initially 40, 70
  - 50 and 60 join concurrently
  - at first 40, 50, and 60 think their successor is 70!
  - which means lookups for 45 will yield 70, not 50
  - after one stabilization, 40 and 50 will learn about 60
  - then 40 will learn about 50

Node Failures

- Assume nodes fail w/o warning (harder issue)
  - Other nodes' routing tables refer to dead node.
  - Dead node's predecessor has no successor.
- If you try to route via dead node, detect timeout, route to numerically closer entry instead.
- Maintain a _list_ of successors: r successors.
  - Lookup answer is first live successor >= key
  - or forward to *any* successor < key

Issues

- How do you characterize the performance of DHTs?
    - How do you improve the performance of DHTs?

Security

- Self-authenticating data, e.g. key = SHA1(value)
  - So DHT node can't forge data, but it is immutable data
- Can someone cause millions of made-up hosts to join? Sybil attack!
  - Can disrupt routing, eavesdrop on all requests, etc.
  - Maybe you can require (and check) that node ID = SHA1(IP address)
- How to deal with route disruptions, storage corruption?
  - Do parallel lookups, replicated store, etc.

CAP Theorem

- Can't have all three of: consistency, availability, tolerance to partitions
- proposed by Eric Brewer in a keynote in 2000
  - later proven by Gilbert & Lynch [2002]
  - but with a specific set of definitions that don't necessarily match what you'd assume (or Brewer meant!)
  - really influential on the design of NoSQL systems
  - and really controversial; “the CAP theorem encourages engineers to make awful decisions.” (Stonebraker)
- usually misinterpreted!

Misinterpretations

- pick any two: consistency, availability, partition tolerance
  - “I want my system to be available, so consistency has to go”
  - or "I need my system to be consistent, so it's not going to be available”
- three possibilities: CP, AP, CA systems

Issues with CAP

- what does it mean to choose or not choose partition tolerance?
  - it's a property of the environment, other two are goals
  - in other words, what's the difference between a "CA" and "CP" system? both give up availability on a partition!
- better phrasing: if the network can have partitions, do we give up on consistency or availability?

Another "P": performance

- providing strong consistency means coordinating across replicas
- besides partitions, also means expensive latency cost
- at least some operations must incur the cost of a wide-area RTT
- can do better with weak consistency: only apply writes locally
  - then propagate asynchronously

CAP Implications

- can't have consistency when:
  - want the system to be always online
  - need to support disconnected operation
  - need faster replies than majority RTT
- in practice: can have consistency and availability together under
  - realistic failure conditions
  - a majority of nodes are up and can communicate
  - can redirect clients to that majority

Dynamo

- Real DHT (1-hop) used inside datacenters
- E.g., shopping cart at Amazon
- More available than Spanner etc.
- Less consistent than Spanner
- Influential — inspired Cassandra

Context

- SLA: 99.9th delay latency < 300ms
- constant failures
- always writeable

Quorums

- Sloppy quorum: first N reachable nodes after the home node on a DHT
- Quorum rule: R + W > N
  - allows you to optimize for the common case
  - but can still provide inconsistencies in the presence of failures (unlike Paxos)

Eventual Consistency

- accept writes at any replica
- allow divergent replicas
- allow reads to see stale or conflicting data
- resolve multiple versions when failures go away
  - latest version if no conflicting updates
  - if conflicts, reader must merge and then write

More Details

- Coordinator: successor of key on a ring
- Coordinator forwards ops to N other nodes on the ring
- Each operation is tagged with the coordinator timestamp
- Values have an associated "vector clock" of coordinator timestamps
- Gets return multiple values along with the vector clocks of values
- Client resolves conflicts and stores the resolved value