Results in DHT Benchmarking

Sean Rhea[†], Timothy Roscoe[§], and John Kubiatowicz[†]
[†]University of California, Berkeley
[§]Intel Research Laboratory at Berkeley
srhea@cs.berkeley.edu, troscoe@intel-research.net, kubitron@cs.berkeley.edu

(Paper in IPTPS 2003: “DHTs Need Application-Driven Benchmarks”)

January 14, 2003

Introduction

- DHT stands for Distributed Hash Table
  - Examples: CAN, Chord, Kademlia, Pastry, Tapestry, . . .

Introduction

- DHT stands for Distributed Hash Table
  - Examples: CAN, Chord, Kademlia, Pastry, Tapestry, ...
- 1000 ft. view: efficiently map user-keys to IP addresses
  - Mostly take $O(\log N)$ time in a network of N nodes

Introduction

- DHT stands for Distributed Hash Table
  - Examples: CAN, Chord, Kademlia, Pastry, Tapestry, ...
- 1000 ft. view: efficiently map user-keys to IP addresses
  - Mostly take $O(\log N)$ time in a network of N nodes
- But this is only a small portion of the story
  - Constant on $\log N$ varies widely
  - Interfaces tuned towards particular usage styles
  - Range of reliability in implementations (and in theory)

Introduction

- DHT stands for Distributed Hash Table
  - Examples: CAN, Chord, Kademlia, Pastry, Tapestry, ...
- 1000 ft. view: efficiently map user-keys to IP addresses
  - Mostly take $O(\log N)$ time in a network of N nodes
- But this is only a small portion of the story
  - Constant on $\log N$ varies widely
  - Interfaces tuned towards particular usage styles
  - Range of reliability in implementations (and in theory)
- As an application developer, there is little data to choose between them
- As a DHT developer, there is no metric for success

Introduction

- DHT stands for Distributed Hash Table
  - Examples: CAN, Chord, Kademlia, Pastry, Tapestry, ...
- 1000 ft. view: efficiently map user-keys to IP addresses
  - Mostly take $O(\log N)$ time in a network of N nodes
- But this is only a small portion of the story
  - Constant on $\log N$ varies widely
  - Interfaces tuned towards particular usage styles
  - Range of reliability in implementations (and in theory)
- As an application developer, there is little data to choose between them
- As a DHT developer, there is no metric for success
- Solution: benchmarking!

Overview

- Algorithm overviews
  - Chord
  - Tapestry
- Towards a common API
- Benchmark results
- Future work

Chord Algorithm

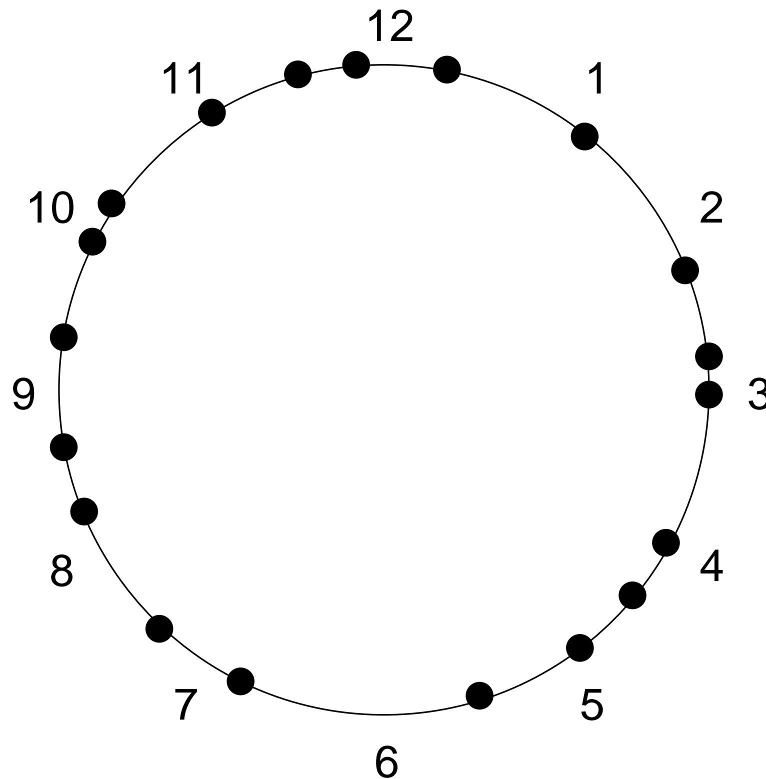
- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$

Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Assign each node an identifier $i \in \mathcal{I}$ (injective mapping, $name : \mathcal{N} \rightarrow \mathcal{I}$)

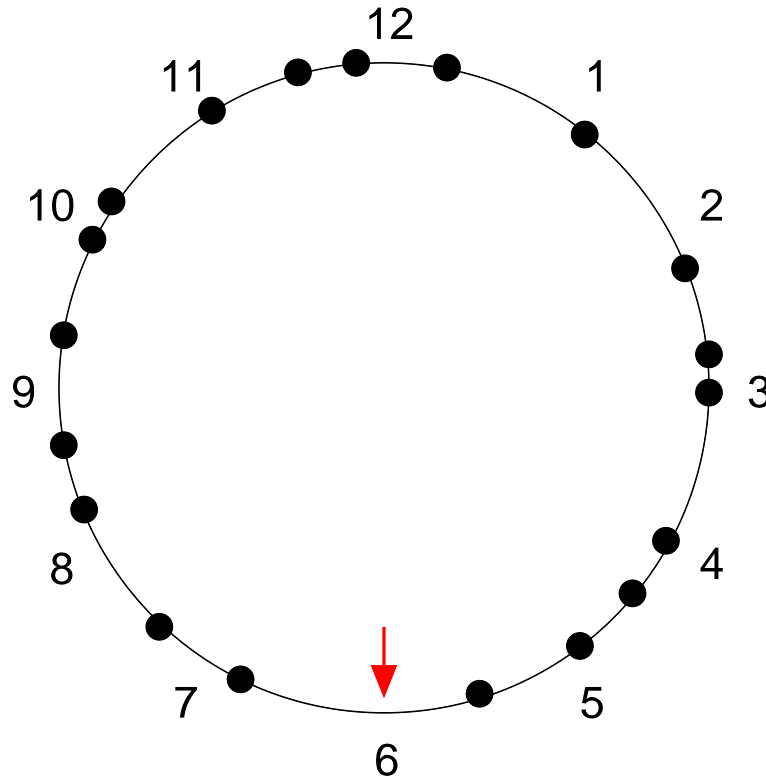
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Assign each node an identifier $i \in \mathcal{I}$ (injective mapping, $name : \mathcal{N} \rightarrow \mathcal{I}$)



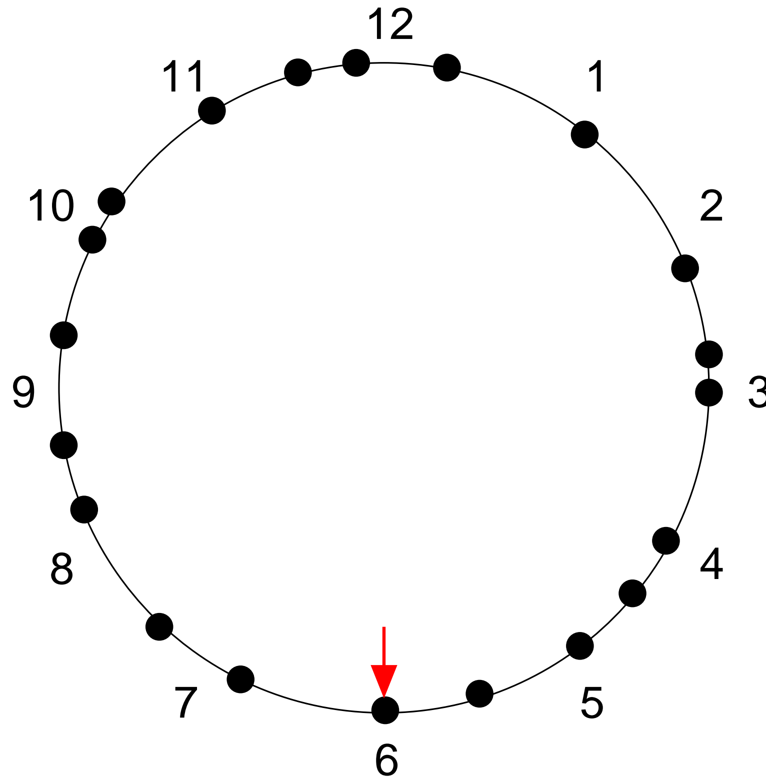
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - To add a new node, update $name$ first



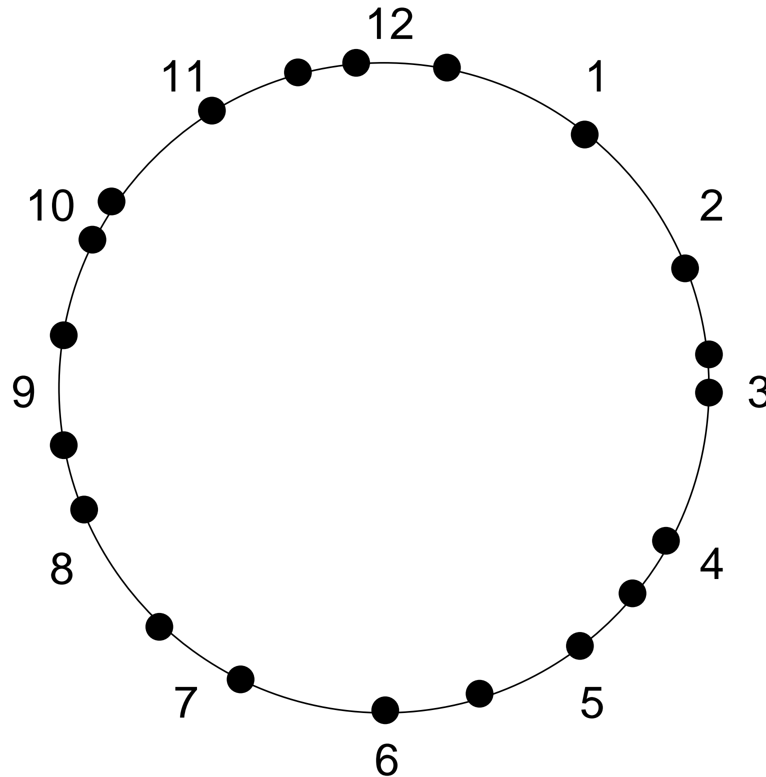
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $\mathit{find_successor} : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Assume we can find the predecessor and successor of our node



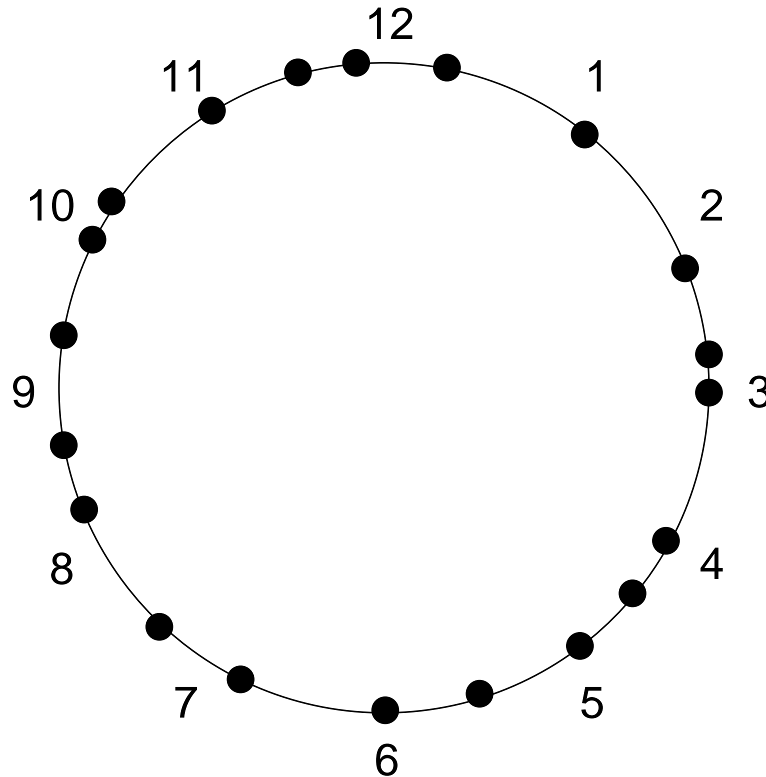
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $\mathit{find_successor} : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Insert ourselves into the ring of nodes



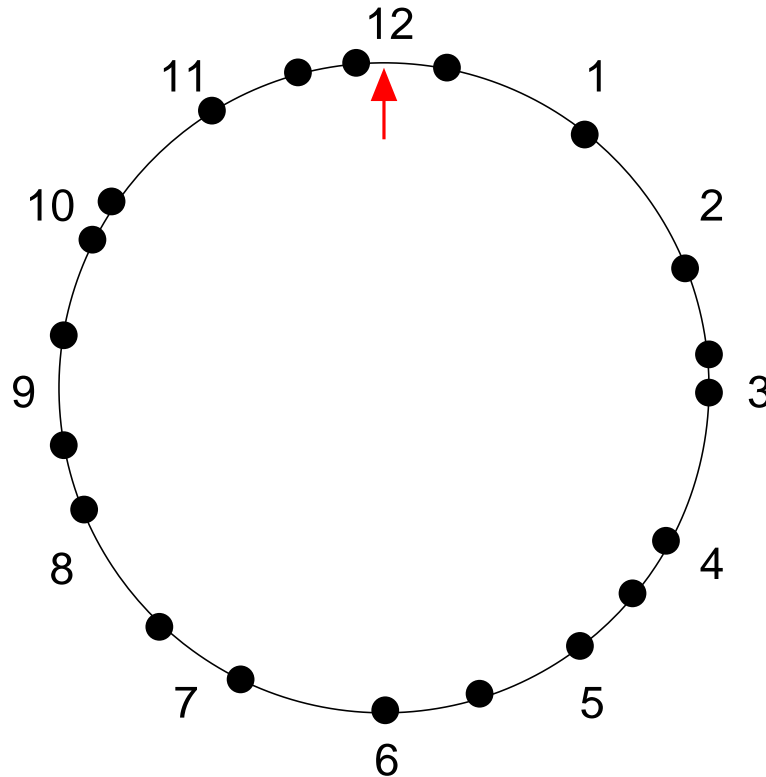
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $\mathit{find_successor} : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Now, how do we evaluate $\mathit{find_successor}$?



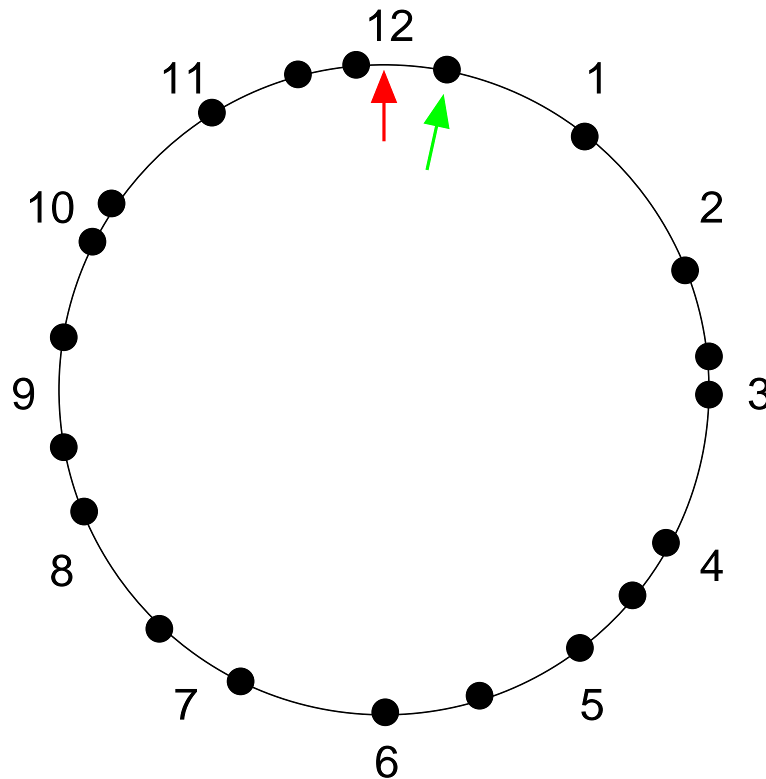
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Starting with an i halfway around $\mathcal{I}$, perform $find_successor(i)$



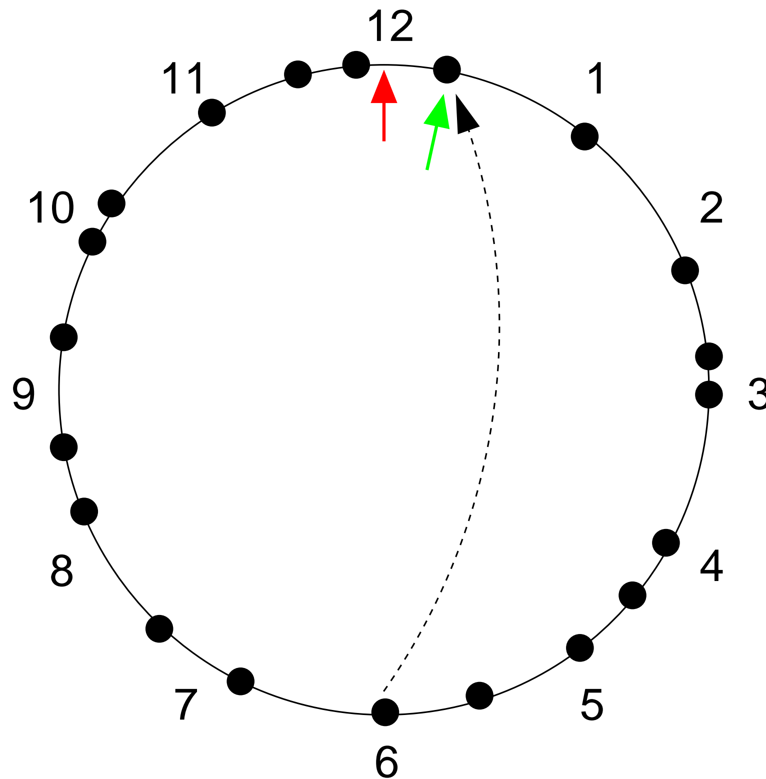
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $\mathit{find_successor} : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Starting with an i halfway around $\mathcal{I}$, perform $\mathit{find_successor}(i)$



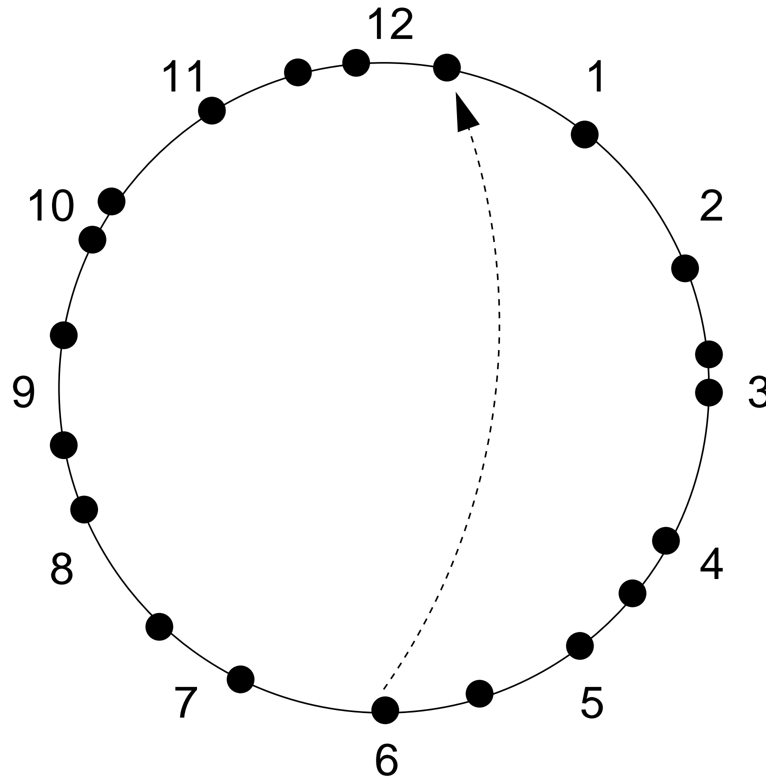
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $\mathit{find_successor} : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Starting with an i halfway around $\mathcal{I}$, perform $\mathit{find_successor}(i)$



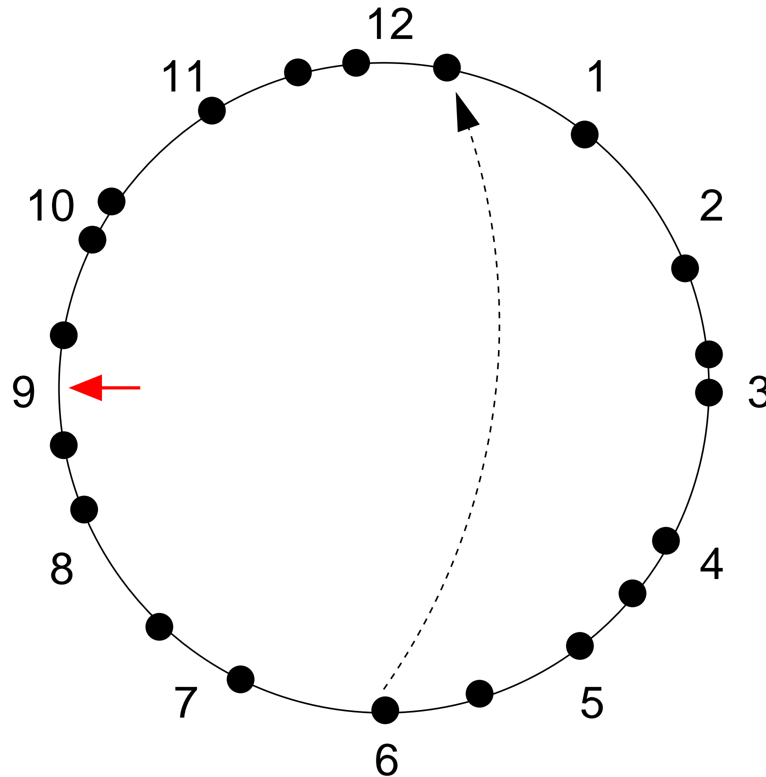
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Remember this node



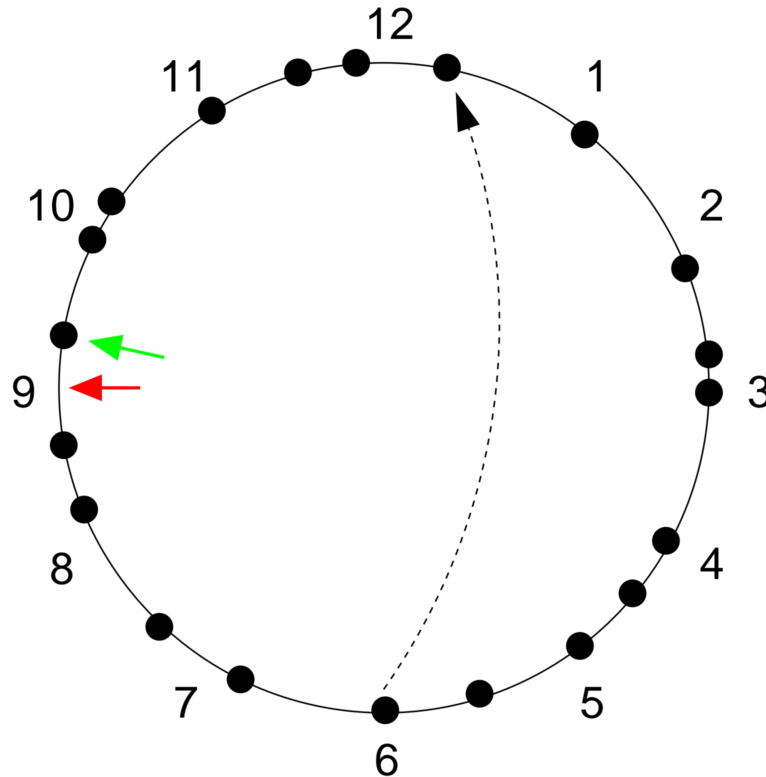
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Repeat this process for i one quarter around $\mathcal{I}$



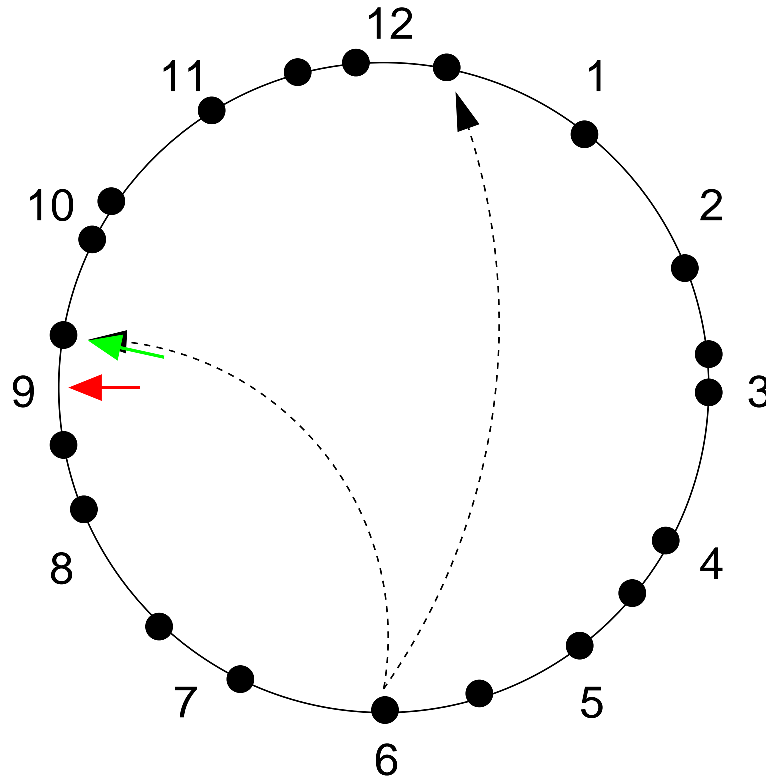
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Repeat this process for i one quarter around $\mathcal{I}$



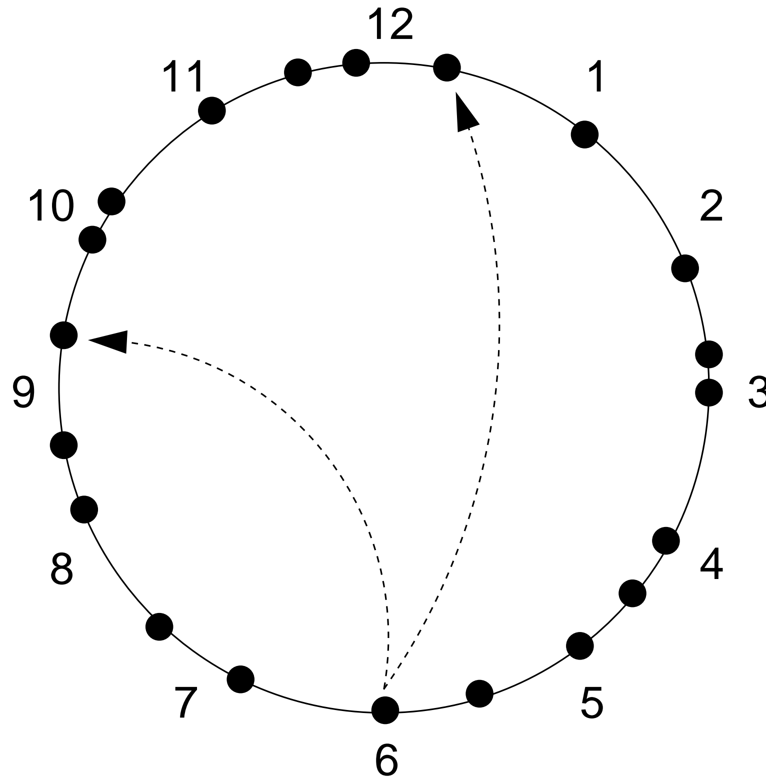
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Repeat this process for i one quarter around $\mathcal{I}$



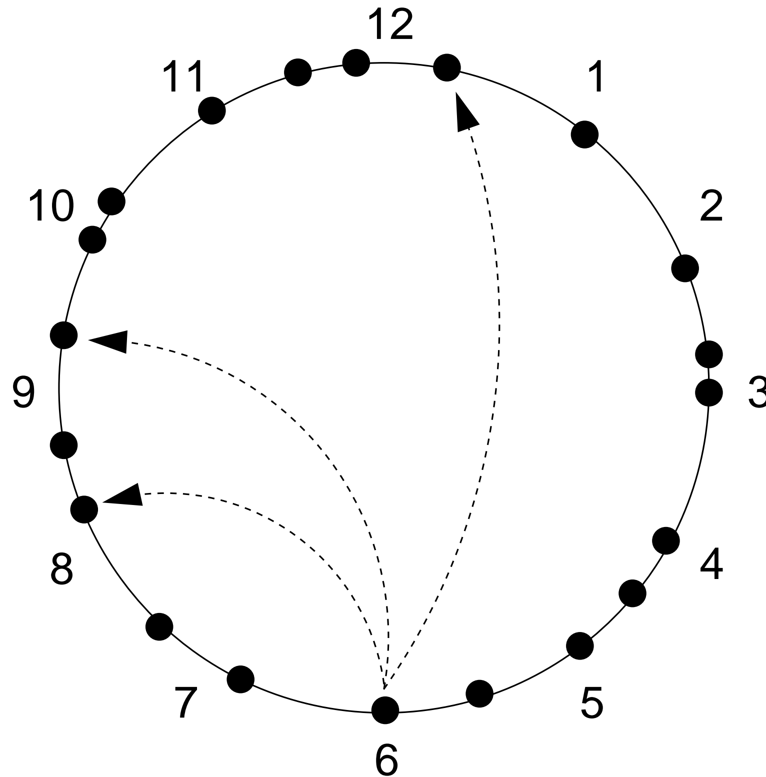
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - And so on. . .



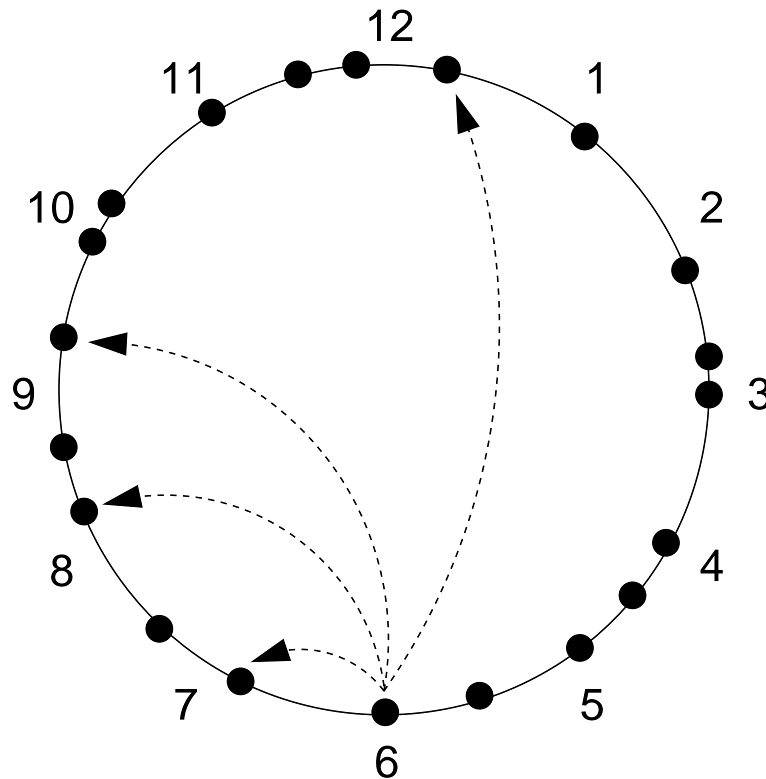
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - And so on. . .



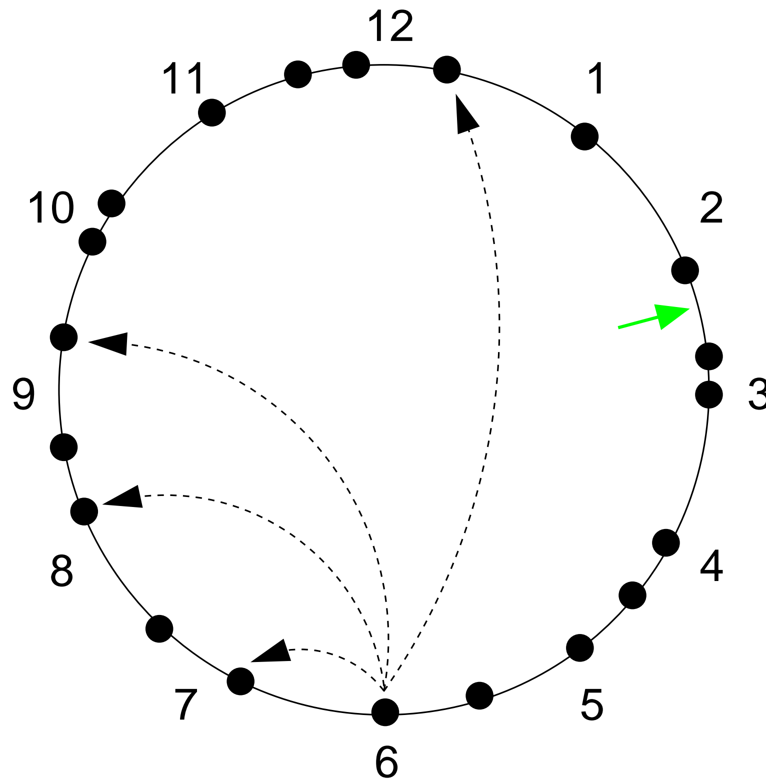
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - And so on. . .until we find our successor node



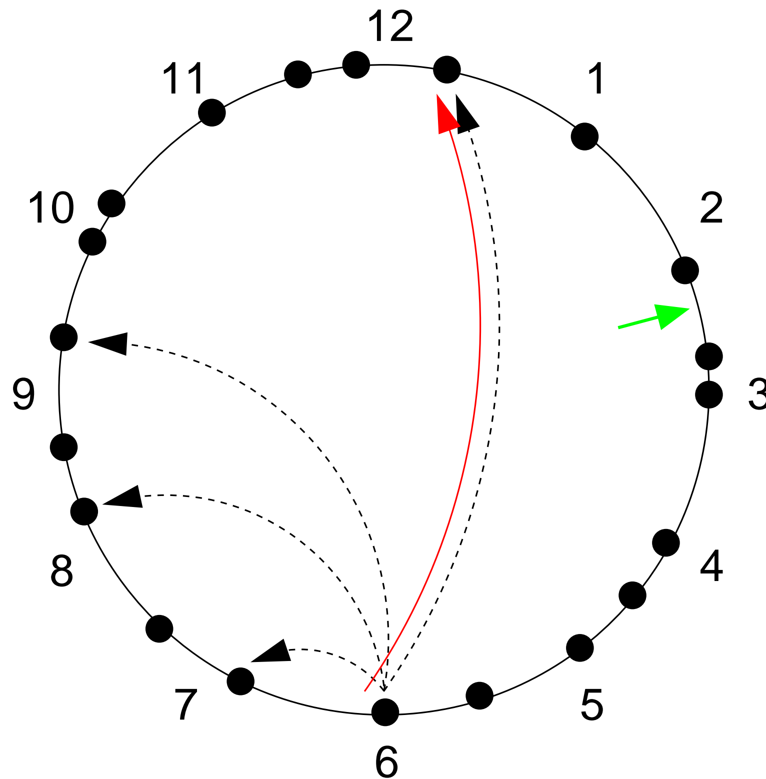
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Now, in order to evaluate $find_successor(i) \ldots$



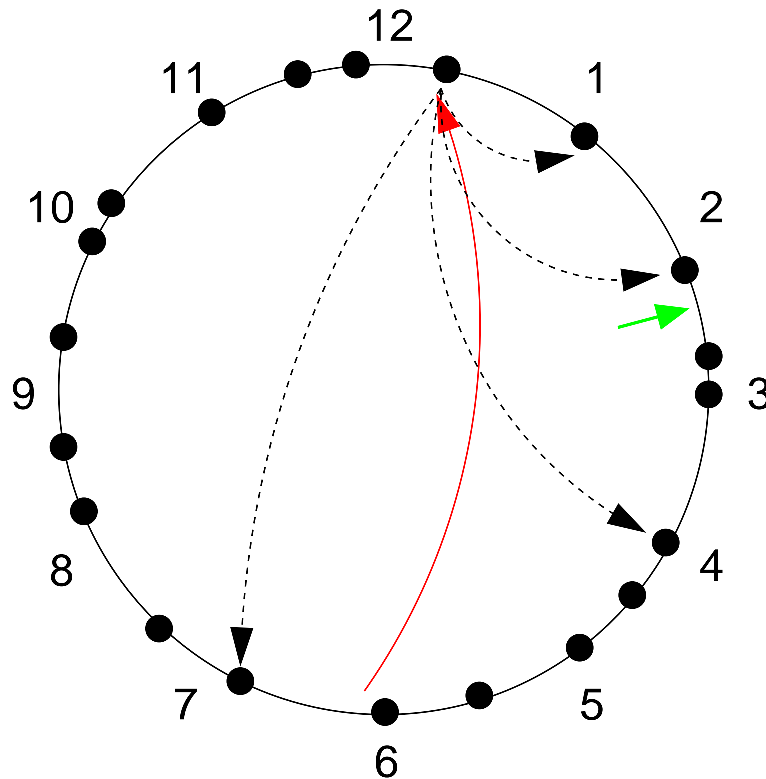
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Now, in order to evaluate $find_successor(i)$, find the node closest to i in $\mathcal{I}$



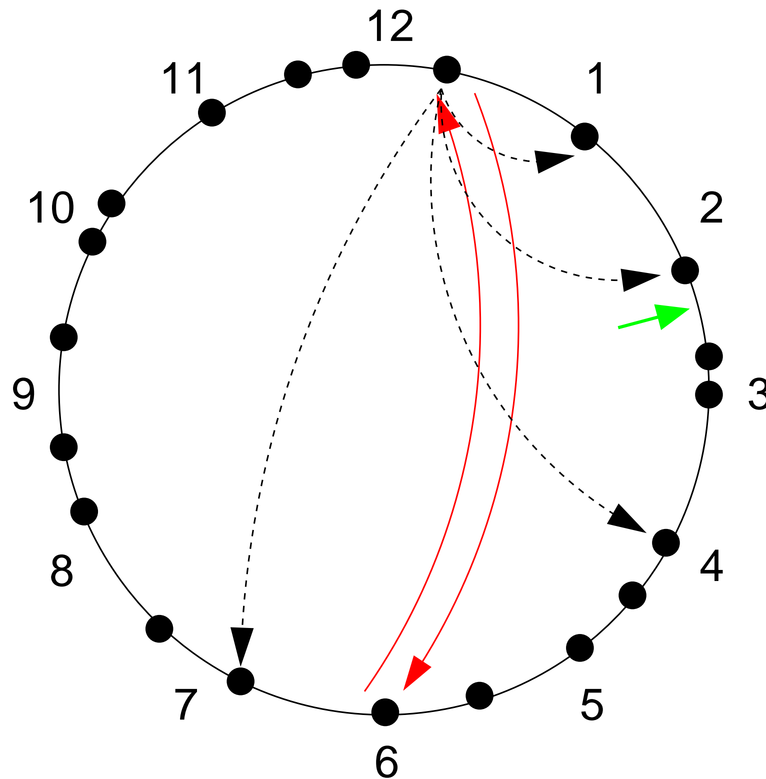
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $\mathit{find_successor} : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Ask it which node it knows of that's closest to i



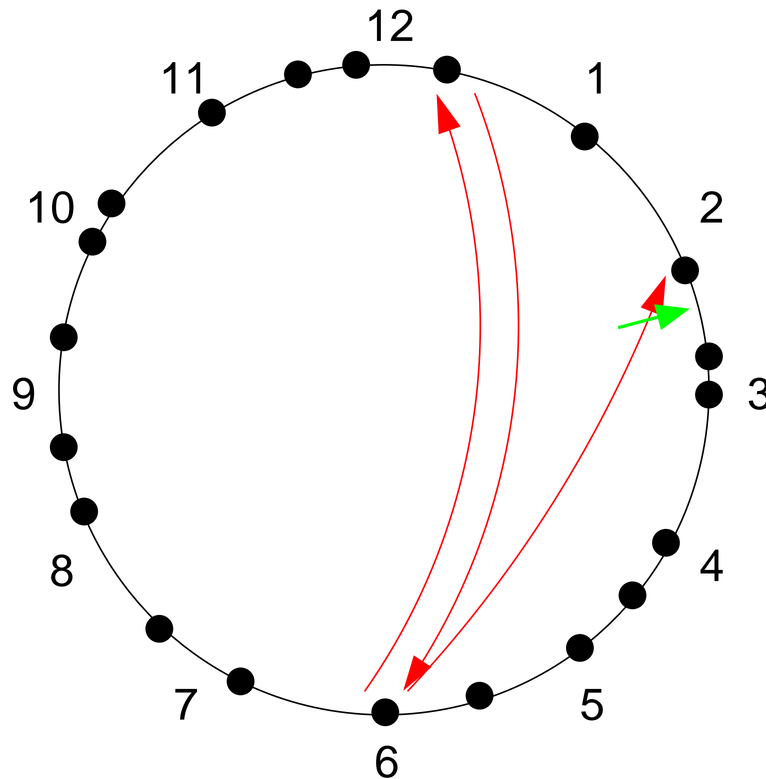
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Ask it which node it knows of that's closest to i



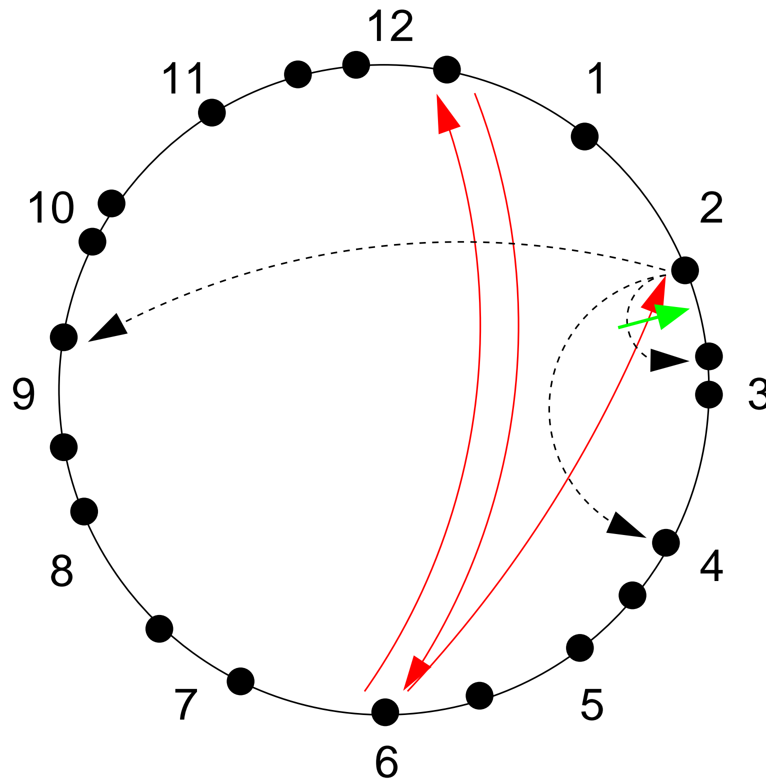
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Then ask that node



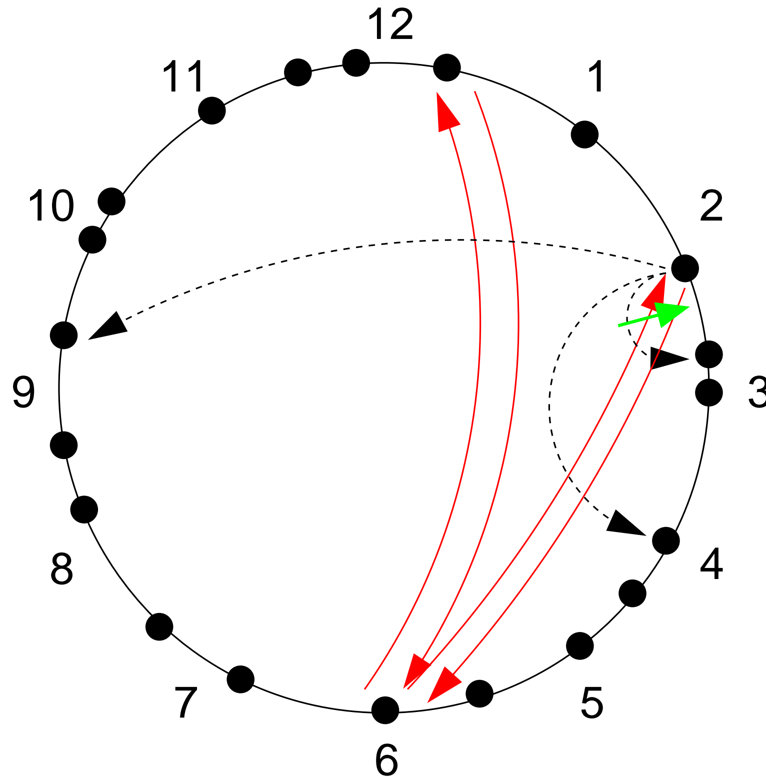
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $find_successor : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - Then ask that node



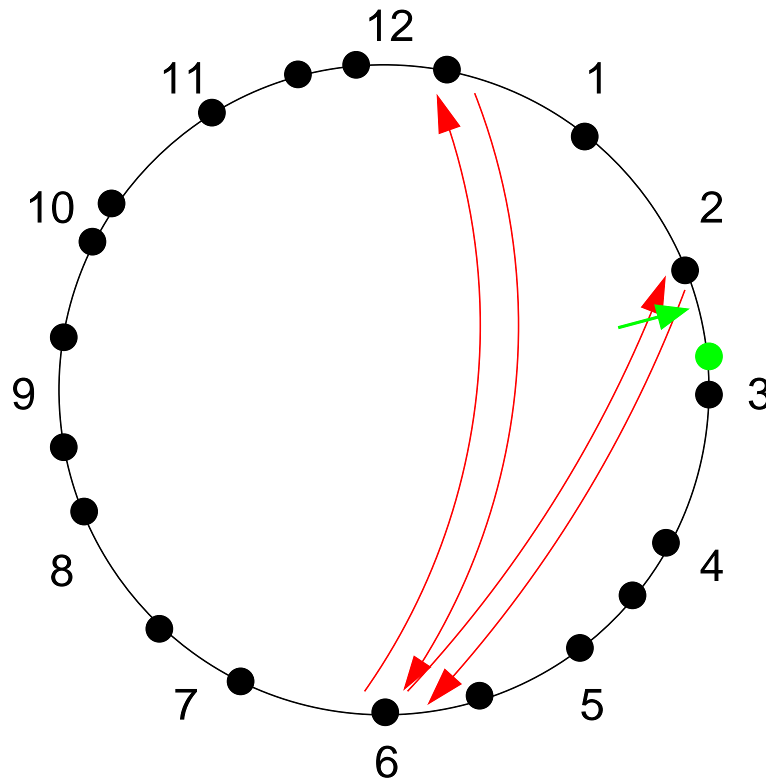
Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $\mathit{find_successor} : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - If i is between $name(n)$ and $name(\mathit{find_successor}(name(n)))\ldots$



Chord Algorithm

- Let $\mathcal{I}$ = 160-bit, circular name space, and $\mathcal{N}$ = a set of nodes
  - Then Chord provides a surjective function, $\mathit{find_successor} : \mathcal{I} \rightarrow \mathcal{N}$
- How?
  - If i is between $name(n)$ and $name(\mathit{find_successor}(name(n)))$, done!



Towards a Common API

- DHTs differ significantly in functionality provided
  - Least common interface contains almost no functionality at all!

Towards a Common API

- DHTs differ significantly in functionality provided
  - Least common interface contains almost no functionality at all!
- Instead, we benchmark the functionality application-developers want
  - Where some function is not implemented by a given DHT, we provide it
  - Examples later

Towards a Common API

- DHTs differ significantly in functionality provided
  - Least common interface contains almost no functionality at all!
- Instead, we benchmark the functionality application-developers want
  - Where some function is not implemented by a given DHT, we provide it
  - Examples later
- Methodology is somewhat unfair
  - Because tested functionality may extend beyond DHTs powers
  - In such cases, performance is subject to quality of our extensions

Towards a Common API

- DHTs differ significantly in functionality provided
  - Least common interface contains almost no functionality at all!
- Instead, we benchmark the functionality application-developers want
  - Where some function is not implemented by a given DHT, we provide it
  - Examples later
- Methodology is somewhat unfair
  - Because tested functionality may extend beyond DHTs powers
  - In such cases, performance is subject to quality of our extensions
  - But this is okay, because:
- Goal is to illustrate the differences between DHTs
  - Not to find a winner
  - In most cases, there are tradeoffs

The Owner Mapping

- Let $\mathcal{I}$ be the set of identifiers in the system
  - Often the set of 160-bit strings

The Owner Mapping

- Let $\mathcal{I}$ be the set of identifiers in the system
  - Often the set of 160-bit strings
- Let $\mathcal{N}$ be the set of nodes in the system
  - Generally a subset of all valid IP:port tuples

The Owner Mapping

- Let $\mathcal{I}$ be the set of identifiers in the system
  - Often the set of 160-bit strings
- Let $\mathcal{N}$ be the set of nodes in the system
  - Generally a subset of all valid IP:port tuples
- Then all current DHTs define a surjective mapping, $owner_{\mathcal{N}} : \mathcal{I} \rightarrow \mathcal{N}$
  - In other words, there is a $n \in \mathcal{N}$ responsible for every $i \in \mathcal{I}$

The Owner Mapping

- Let $\mathcal{I}$ be the set of identifiers in the system
  - Often the set of 160-bit strings
- Let $\mathcal{N}$ be the set of nodes in the system
  - Generally a subset of all valid IP:port tuples
- Then all current DHTs define a surjective mapping, $owner_{\mathcal{N}} : \mathcal{I} \rightarrow \mathcal{N}$
  - In other words, there is a $n \in \mathcal{N}$ responsible for every $i \in \mathcal{I}$
  - In Tapestry, $owner_{\mathcal{N}}(i)$ is called the *root* of i
  - In Chord, $owner_{\mathcal{N}}(i)$ is called the *successor* of i

The Owner Mapping

- Let $\mathcal{I}$ be the set of identifiers in the system
  - Often the set of 160-bit strings
- Let $\mathcal{N}$ be the set of nodes in the system
  - Generally a subset of all valid IP:port tuples
- Then all current DHTs define a surjective mapping, $owner_{\mathcal{N}} : \mathcal{I} \rightarrow \mathcal{N}$
  - In other words, there is a $n \in \mathcal{N}$ responsible for every $i \in \mathcal{I}$
  - In Tapestry, $owner_{\mathcal{N}}(i)$ is called the *root* of i
  - In Chord, $owner_{\mathcal{N}}(i)$ is called the *successor* of i
- Note that $owner$ is parameterized on $\mathcal{N}$
  - In particular, adding nodes to $\mathcal{N}$ changes the mapping

Interfaces to the Owner Mapping

- The $owner$ mapping is exposed in a variety of ways
  - Chord provides a function $find_owner(i)$ (called $find_successor$ earlier)
    - similar to a DNS lookup

Interfaces to the Owner Mapping

- The $owner$ mapping is exposed in a variety of ways
  - Chord provides a function $find_owner(i)$ (called $find_successor$ earlier)
    - similar to a DNS lookup
  - Tapestry provides a function $call_owner(i, m)$ (called a *route* message)
    - sends a message m to $owner_{\mathcal{N}}(i)$

Interfaces to the Owner Mapping

- The $owner$ mapping is exposed in a variety of ways
  - Chord provides a function $find_owner(i)$ (called $find_successor$ earlier)
    - similar to a DNS lookup
  - Tapestry provides a function $call_owner(i, m)$ (called a *route* message)
    - sends a message m to $owner_{\mathcal{N}}(i)$
- Can implement each using other
  - $call_owner$ in Chord is a $find_owner$ plus one network message
  - $find_owner$ in Tapestry is a $call_owner$ plus one network message

Interfaces to the Owner Mapping

- The $owner$ mapping is exposed in a variety of ways
  - Chord provides a function $find_owner(i)$ (called $find_successor$ earlier)
    - similar to a DNS lookup
  - Tapestry provides a function $call_owner(i, m)$ (called a *route* message)
    - sends a message m to $owner_{\mathcal{N}}(i)$
- Can implement each using other
  - $call_owner$ in Chord is a $find_owner$ plus one network message
  - $find_owner$ in Tapestry is a $call_owner$ plus one network message
- Some DHTs also expose $owner^{-1}$
  - Chord notifies applications running on n when $owner_{\mathcal{N}}^{-1}(n)$ changes

The Locate Object Function

- Often DHTs are used to store and retrieve objects
  - As opposed to locating computational services, for example

The Locate Object Function

- Often DHTs are used to store and retrieve objects
  - As opposed to locating computational services, for example
- DHASH is a storage layer build on Chord
  - $put(x)$ stores an object x on $owner_{\mathcal{N}}(SHA(x))$
  - $get(i)$ retrieves an object x s.t. $i = SHA(x)$ from $owner_{\mathcal{N}}(SHA(x))$

The Locate Object Function

- Often DHTs are used to store and retrieve objects
  - As opposed to locating computational services, for example
- DHASH is a storage layer build on Chord
  - $put(x)$ stores an object x on $owner_{\mathcal{N}}(SHA(x))$
  - $get(i)$ retrieves an object x s.t. $i = SHA(x)$ from $owner_{\mathcal{N}}(SHA(x))$
- Tapestry has no storage layer
  - Instead, a node storing x can $publish(SHA(x))$
    * Stores the *location* of x in DHT, not x itself

The Locate Object Function

- Often DHTs are used to store and retrieve objects
  - As opposed to locating computational services, for example
- DHASH is a storage layer build on Chord
  - $put(x)$ stores an object x on $owner_{\mathcal{N}}(SHA(x))$
  - $get(i)$ retrieves an object x s.t. $i = SHA(x)$ from $owner_{\mathcal{N}}(SHA(x))$
- Tapestry has no storage layer
  - Instead, a node storing x can $publish(SHA(x))$
    - Stores the *location* of x in DHT, not x itself
  - Then, interested nodes can $call_obj(SHA(x), m)$
    - Sends a message m to some node which has published $SHA(x)$
    - Probabilistically guaranteed to be the closest such node

The Locate Object Function

- Often DHTs are used to store and retrieve objects
  - As opposed to locating computational services, for example
- DHASH is a storage layer build on Chord
  - $put(x)$ stores an object x on $owner_{\mathcal{N}}(SHA(x))$
  - $get(i)$ retrieves an object x s.t. $i = SHA(x)$ from $owner_{\mathcal{N}}(SHA(x))$
- Tapestry has no storage layer
  - Instead, a node storing x can $publish(SHA(x))$
    * Stores the *location* of x in DHT, not x itself
  - Then, interested nodes can $call_obj(SHA(x), m)$
    * Sends a message m to some node which has published $SHA(x)$
    * Probabilistically guaranteed to be the closest such node
- Again, can implement each using other

Our Common API

- We chose the following functions as most important to applications
  - $join(g)$ — join a network $\mathcal{N}$ where $g \in \mathcal{N}$
  - $leave$ — leave the network
  - $find_owner(i)$ — find $owner_{\mathcal{N}}(i)$
  - $call_owner(i, m)$ — send message m to $owner_{\mathcal{N}}(i)$
  - $find_obj(i)$ — find a node storing an object named i
  - $call_obj(i, m)$ — send message m to a node storing an object named i
  - $retrieve_obj(i)$ — find and retrieve an object named i

Our Common API

- We chose the following functions as most important to applications
  - $join(g)$ — join a network $\mathcal{N}$ where $g \in \mathcal{N}$
  - $leave$ — leave the network
  - $find_owner(i)$ — find $owner_{\mathcal{N}}(i)$
  - $call_owner(i, m)$ — send message m to $owner_{\mathcal{N}}(i)$
  - $find_obj(i)$ — find a node storing an object named i
  - $call_obj(i, m)$ — send message m to a node storing an object named i
  - $retrieve_obj(i)$ — find and retrieve an object named i
- Not quite sure yet what to do with $publish$ and put
  - Don't really fit
  - In Tapestry, $publish(i)$ is a $call_owner(i, m)$ with a small m
  - In Chord, $put(x)$ is a $call_owner(SHA(x), x)$

Our Common API

- We chose the following functions as most important to applications
  - $join(g)$ — join a network $\mathcal{N}$ where $g \in \mathcal{N}$
  - $leave$ — leave the network
  - $find_owner(i)$ — find $owner_{\mathcal{N}}(i)$
  - $call_owner(i, m)$ — send message m to $owner_{\mathcal{N}}(i)$
  - $find_obj(i)$ — find a node storing an object named i
  - $call_obj(i, m)$ — send message m to a node storing an object named i
  - $retrieve_obj(i)$ — find and retrieve an object named i
- Not quite sure yet what to do with $publish$ and put
  - Don't really fit
  - In Tapestry, $publish(i)$ is a $call_owner(i, m)$ with a small m
  - In Chord, $put(x)$ is a $call_owner(SHA(x), x)$
- In this talk, I will focus on $find_owner$ and $find_obj$

Experimental Setup

- All experiments performed on PlanetLab
  - Research network spread throughout US, Europe, and Australia
  - Total of 83 nodes used in tests, up to 3 nodes per site
  - Mostly 1.2 GHz CPUs with 1 GB of RAM

Experimental Setup

- All experiments performed on PlanetLab
  - Research network spread throughout US, Europe, and Australia
  - Total of 83 nodes used in tests, up to 3 nodes per site
  - Mostly 1.2 GHz CPUs with 1 GB of RAM
- Used implementations supplied by algorithm designers
  - For Chord, used the version from MIT (discussed in the CFS paper)
  - For Tapestry, used the OceanStore implementation

Experimental Setup

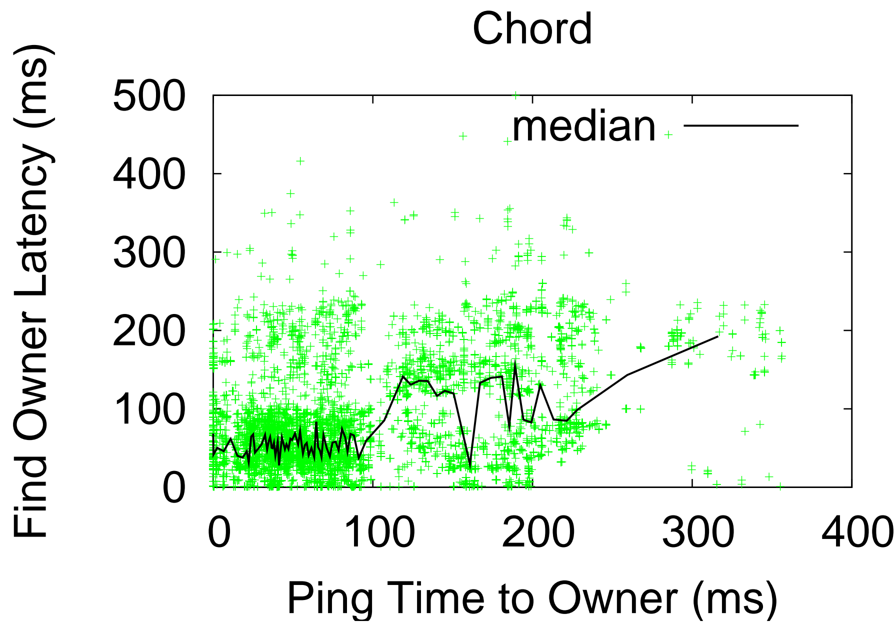
- All experiments performed on PlanetLab
  - Research network spread throughout US, Europe, and Australia
  - Total of 83 nodes used in tests, up to 3 nodes per site
  - Mostly 1.2 GHz CPUs with 1 GB of RAM
- Used implementations supplied by algorithm designers
  - For Chord, used the version from MIT (discussed in the CFS paper)
  - For Tapestry, used the OceanStore implementation
  - Want to encourage quality implementations, as well as designs

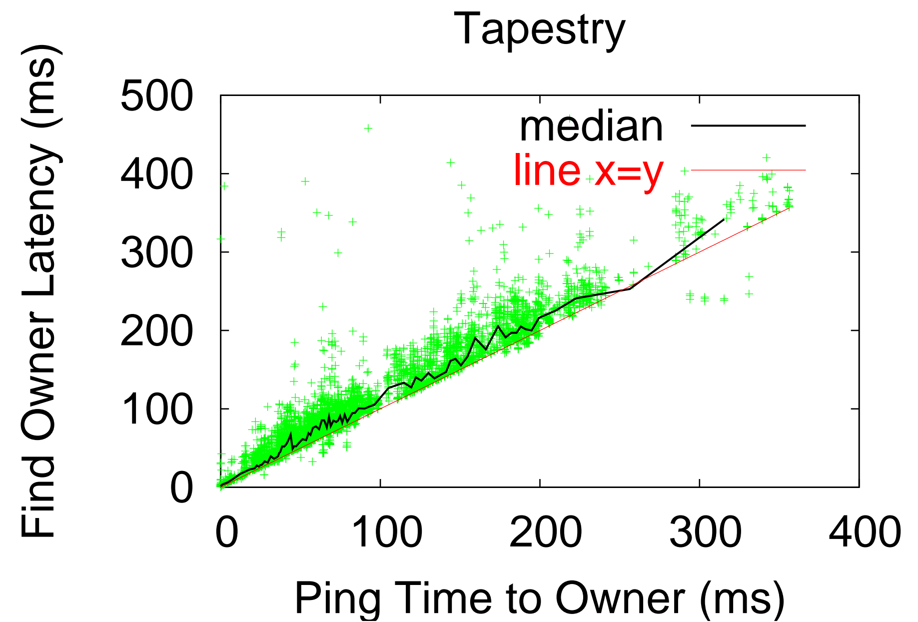
Experimental Setup

- All experiments performed on PlanetLab
  - Research network spread throughout US, Europe, and Australia
  - Total of 83 nodes used in tests, up to 3 nodes per site
  - Mostly 1.2 GHz CPUs with 1 GB of RAM
- Used implementations supplied by algorithm designers
  - For Chord, used the version from MIT (discussed in the CFS paper)
  - For Tapestry, used the OceanStore implementation
  - Want to encourage quality implementations, as well as designs
- All experiments run with a constant $\mathcal{N}$
  - Bring the network up, allow it to stabilize, then run experiments

Find Owner Results

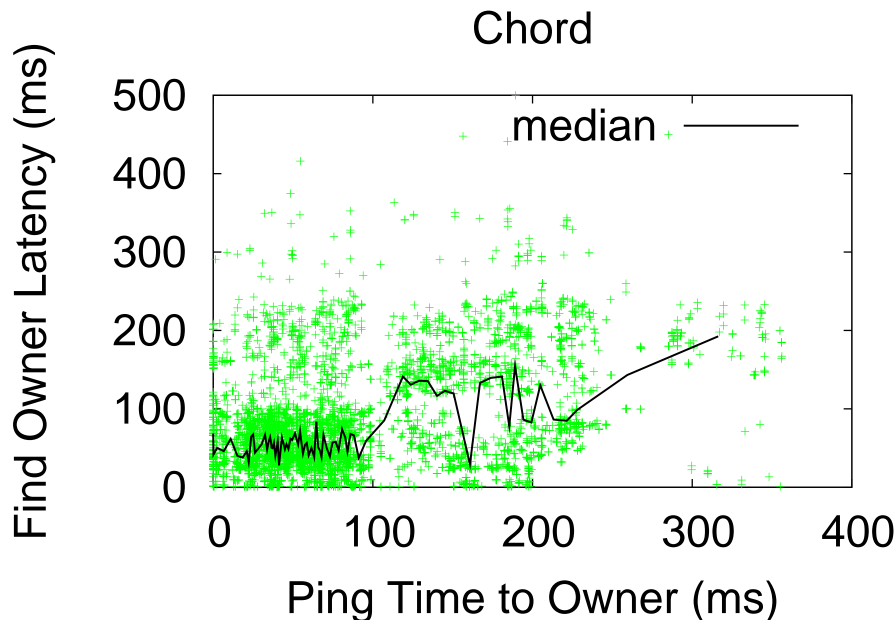
- Chord $find_owner$ time roughly independent of ping time to $owner$
  - Somewhat dependent because of non-uniform node distribution
- Tapestry $find_owner$ time roughly follows ping time to $owner$
  - As predicted by theory
- Chord median is 62.3 ms, Tapestry median is 85.2 ms

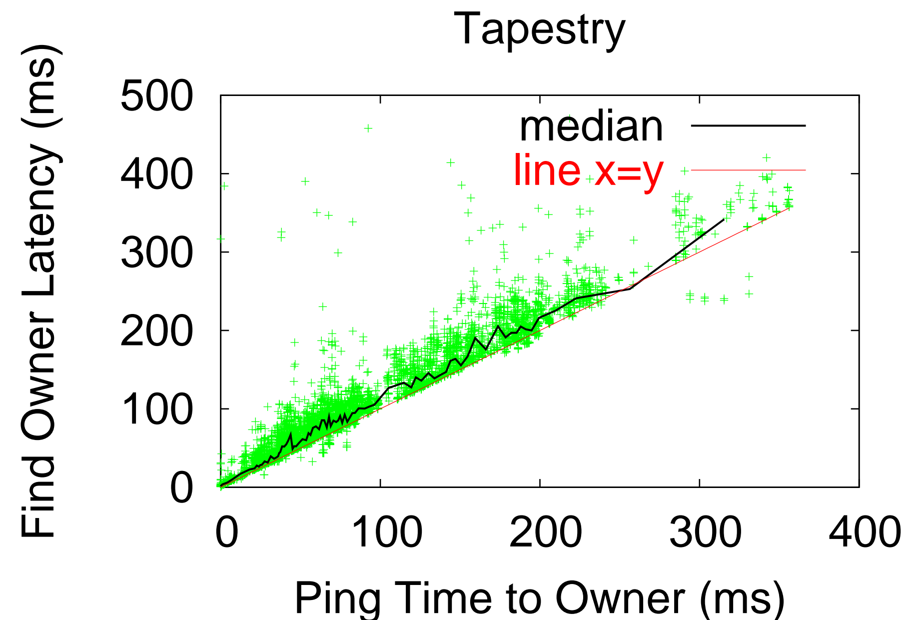




Find Owner Results

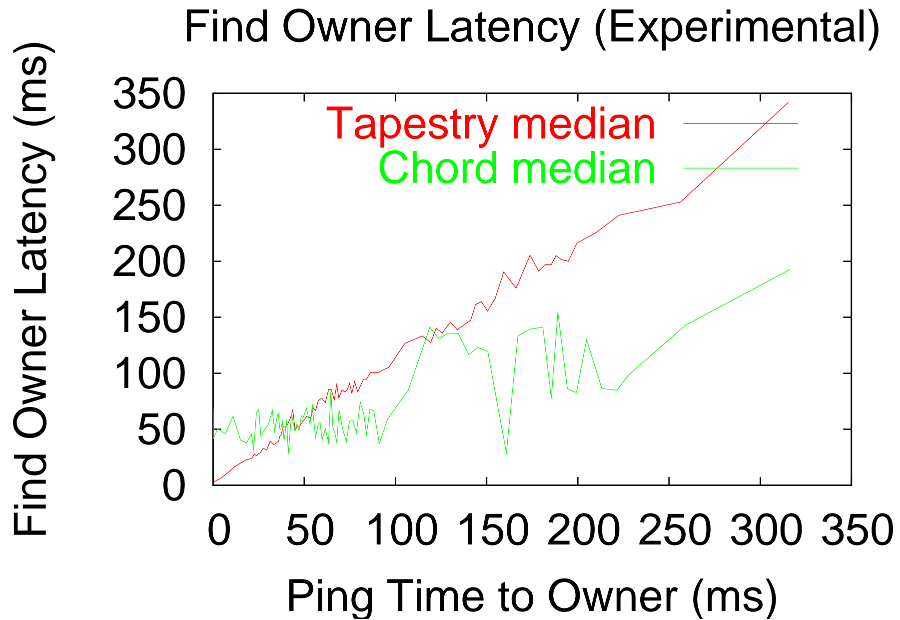
- Chord *find_owner* time roughly independent of ping time to *owner*
  - Somewhat dependent because of non-uniform node distribution
- Tapestry *find_owner* time roughly follows ping time to *owner*
  - As predicted by theory
- Chord median is 62.3 ms, Tapestry median is 85.2 ms
  - Median internode ping time in PlanetLab is 64.9 ms!





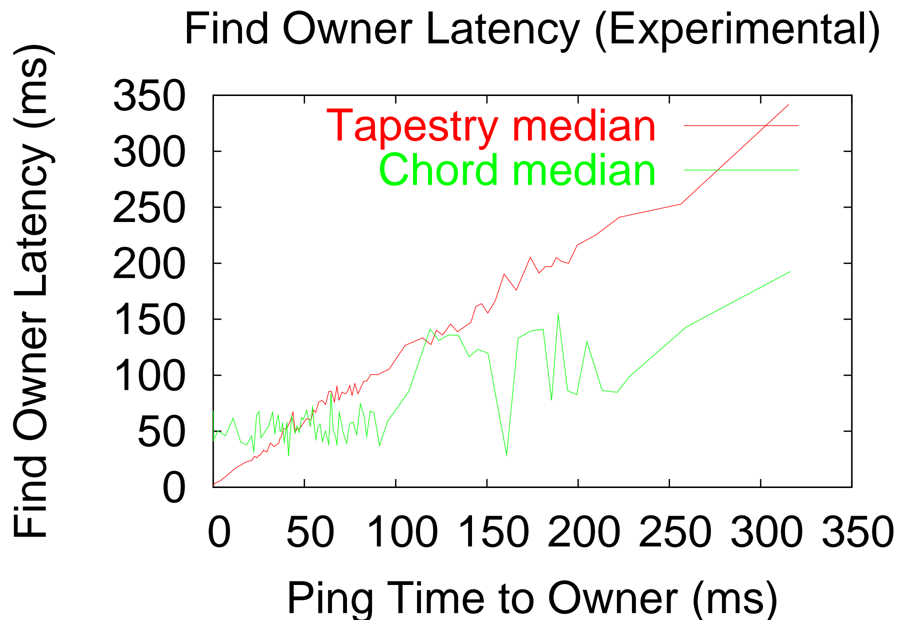
Find Owner Results (con't.)

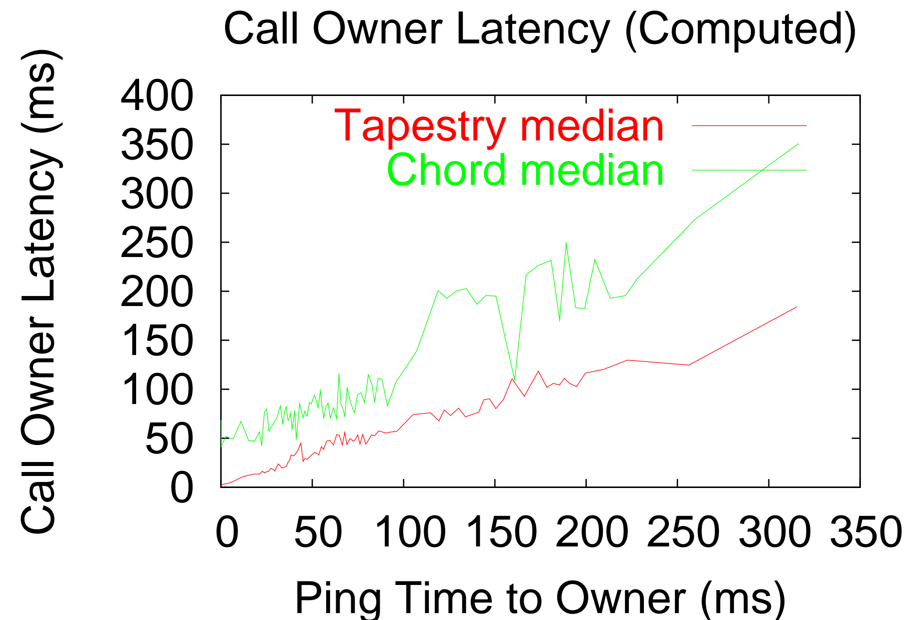
- Chord is often faster for $find_owner$
  - New Chord routing algorithm is very effective



Find Owner Results (con't.)

- Chord is often faster for $find_owner$
  - New Chord routing algorithm is very effective
- Can use $find_owner$ times to estimate $call_owner$ ones
  - Tapestry $call_owner$ time is one message faster than $find_owner$
  - Chord $call_owner$ time is one message slower than $find_owner$





Find Object Benchmark

- In DHASH, backup replicas are stored on successors of $owner$
  - Convenient for fault-tolerance

Find Object Benchmark

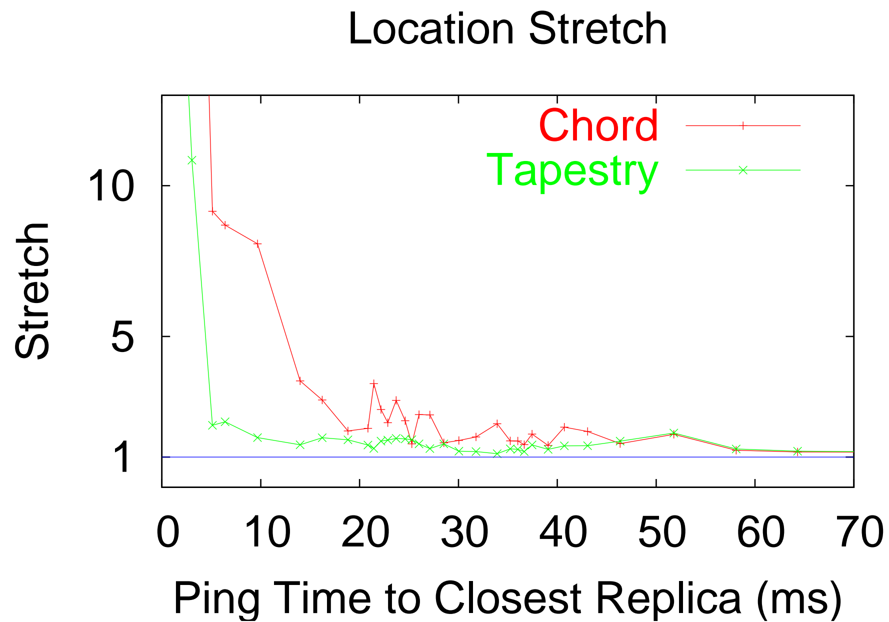
- In DHASH, backup replicas are stored on successors of $owner$
  - Convenient for fault-tolerance
- For $find_obj$ benchmark, store 4 replicas each of 10 different objects
  - Using DHASH replication strategy
  - Then every node in the system does a $find_obj$ on each object
  - Store replicas on same nodes when testing Tapestry

Find Object Benchmark

- In DHASH, backup replicas are stored on successors of $owner$
  - Convenient for fault-tolerance
- For $find_obj$ benchmark, store 4 replicas each of 10 different objects
  - Using DHASH replication strategy
  - Then every node in the system does a $find_obj$ on each object
  - Store replicas on same nodes when testing Tapestry
- Metric is the *quality* of location
  - How close to the query source is the discovered replica
  - Versus how close is the closest replica
- Also note the location time

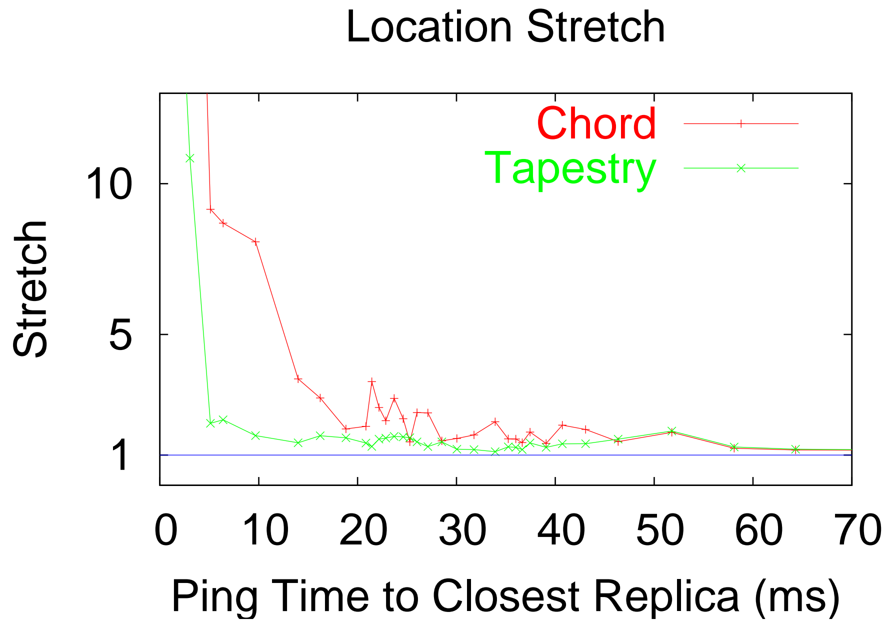
Find Object Results

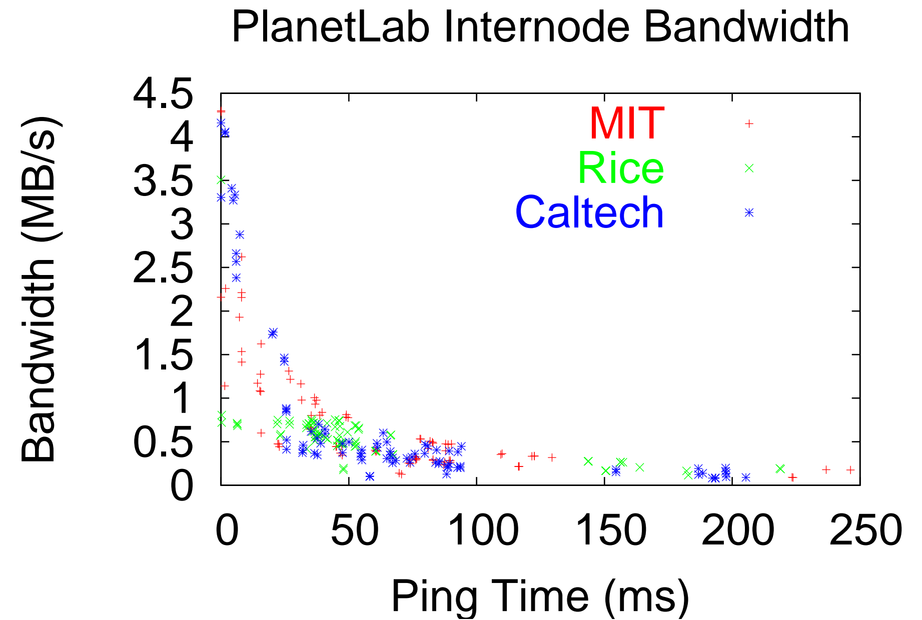
- Median ping time to discovered replica 54.5 ms in Chord, 39.1 ms in Tapestry
  - 28% latency reduction



Find Object Results

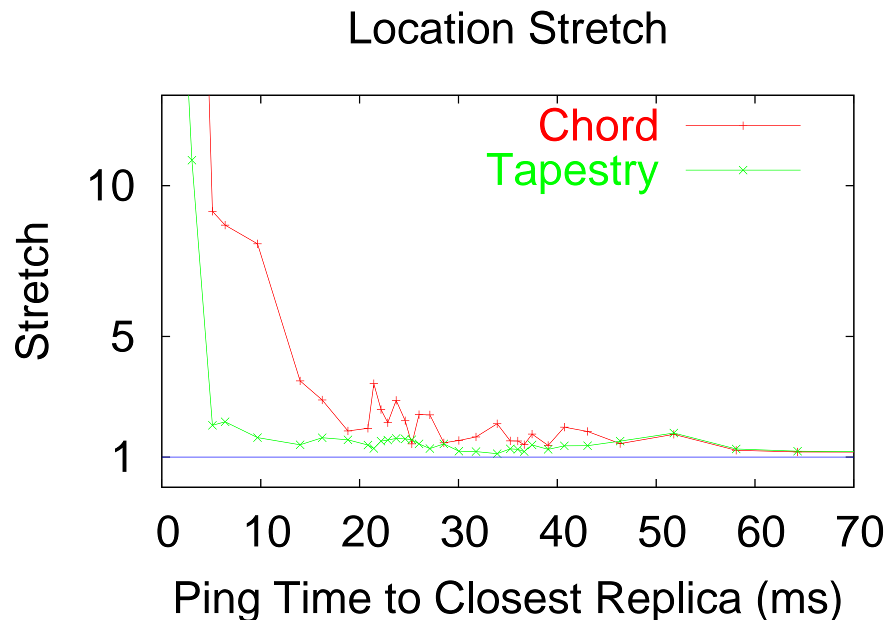
- Median ping time to discovered replica 54.5 ms in Chord, 39.1 ms in Tapestry
  - 28% latency reduction, 47% bandwidth improvement to discovered replica
  - Associated bandwidths: 449 kB/s vs. 661 kB/s

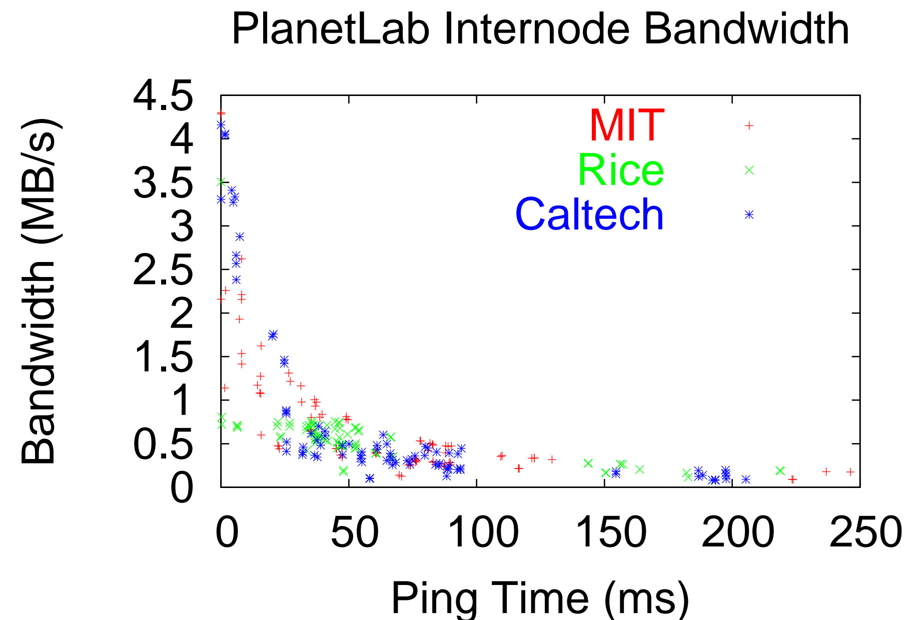




Find Object Results

- Median ping time to discovered replica 54.5 ms in Chord, 39.1 ms in Tapestry
  - 28% latency reduction, 47% bandwidth improvement to discovered replica
  - Associated bandwidths: 449 kB/s vs. 661 kB/s
- Median $find_obj$ times were 60.5 ms in Chord and 64.7 ms in Tapestry
  - Median $call_obj$ times computed as 87.8 ms and 45.2 ms





Future Work

- Test more functionality!
  - Actual tests for $call_owner$, $call_obj$, $retrieve_obj$
  - As opposed to calculations based on $find_owner$, $find_obj$
  - Reliability benchmarks

Future Work

- Test more functionality!
  - Actual tests for $call_owner$, $call_obj$, $retrieve_obj$
  - As opposed to calculations based on $find_owner$, $find_obj$
  - Reliability benchmarks
- Test more algorithms
  - Currently have Chord and Tapestry wrapped in a common framework
  - Working on Pastry, CAN, and routing-style (as opposed to DNS-style) Chord

Future Work

- Test more functionality!
  - Actual tests for $call_owner$, $call_obj$, $retrieve_obj$
  - As opposed to calculations based on $find_owner$, $find_obj$
  - Reliability benchmarks
- Test more algorithms
  - Currently have Chord and Tapestry wrapped in a common framework
  - Working on Pastry, CAN, and routing-style (as opposed to DNS-style) Chord
- Simulation and Emulation
  - Use of PlanetLab gives reality, but still a small network
  - Plan to use Emulab—subject to poor topology choice, but larger networks
  - Also plan to use simulation—no computation costs, but *very* large networks

Future Work

- Test more functionality!
  - Actual tests for $call_owner$, $call_obj$, $retrieve_obj$
  - As opposed to calculations based on $find_owner$, $find_obj$
  - Reliability benchmarks
- Test more algorithms
  - Currently have Chord and Tapestry wrapped in a common framework
  - Working on Pastry, CAN, and routing-style (as opposed to DNS-style) Chord
- Simulation and Emulation
  - Use of PlanetLab gives reality, but still a small network
  - Plan to use Emulab—subject to poor topology choice, but larger networks
  - Also plan to use simulation—no computation costs, but *very* large networks
- Remember the goal
  - Give application designers information they need to choose a DHT
  - Give DHT designers metrics for success