Assignment 2

Implementing Chord

Distributed Hash Tables (DHT)

Motivation: More efficient search in P2P networks

Idea: Hash tables offer key/value-mapping

Principle: Every peer and object has a unique name

- Calculate a hash function on unique name
- Peers and objects map onto points in hash-space
- Peer “closest” to object is responsible for that object

Many different research projects on this topic

- Hash function, hash-space, and metric differ, but…
- Examples: Chord (MIT), CAN (UC Berkeley), Pastry (Microsoft & Rice Univ.), Tapestry (UC Berkeley)

DHT as basis for building more complex services

DHT Example: Chord

Chord uses SHA-1 hash function ⇨ 160 bit ID

ID maps on to identifier circle (modulo 2^{160})

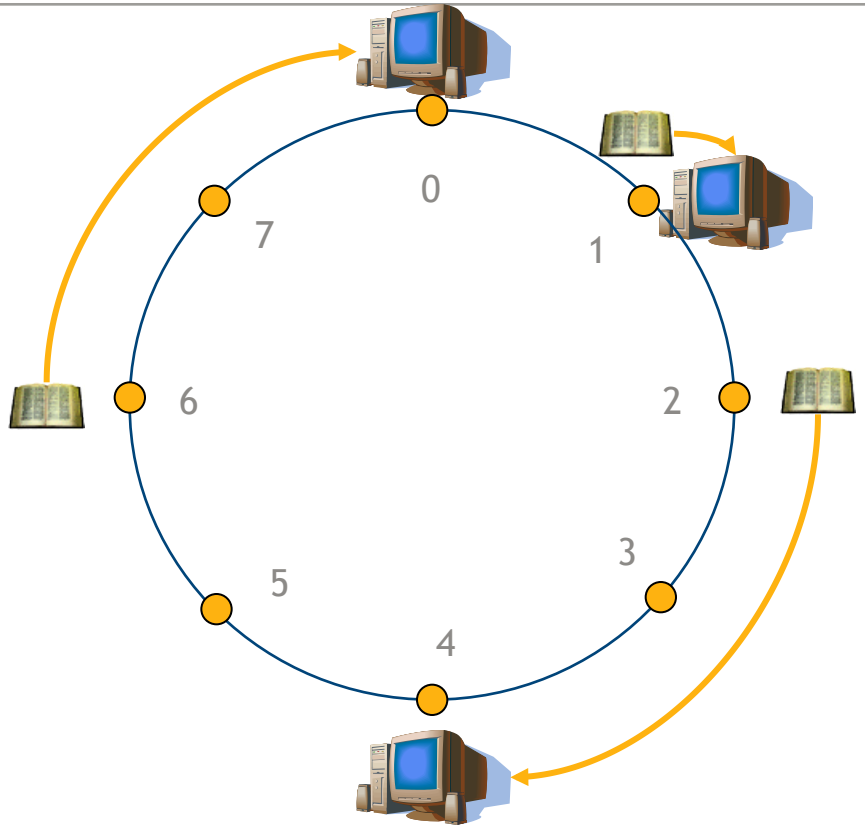
Node responsible for keys with IDs that precede it

Example: 3-bit ID circle

Nodes keep track of their predecessor and successor nodes

Routing by passing query to successor

- Not good for large networks
- Finger tables solve that problem



Goal of the Assignment

Implement basic Chord

Simple application

- Distributed data storage

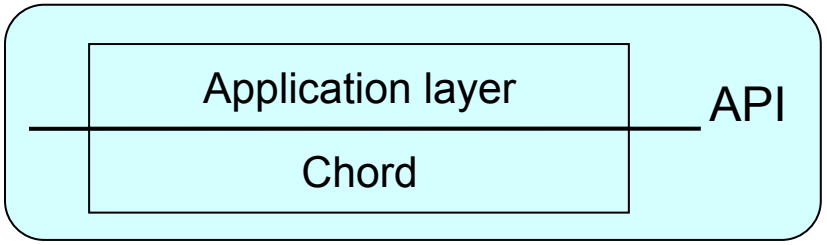
Chord API

Chord ring construction and maintenance

Request passing along the ring

Application and Chord

Application sits on top of Chord



API has 4 functions:

- Join – join the network
- Leave – leave the network
- Store – store a key-value pair
- Retrieve – retrieve a stored value

Application

Simple application with a simple interface

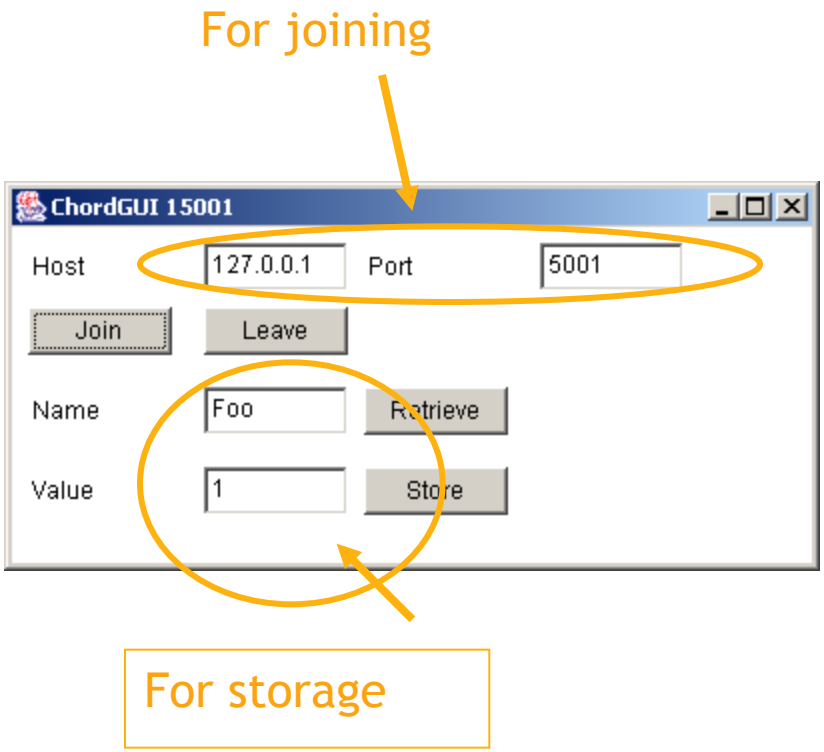
- Graphical or text, as long as it works

Application stores values in Chord

- Names are strings, keys integers, and values integers

Application calls API through calls defined in assignment sheet (with the given parameters)

Application should give feedback to user (text or GUI)



Chord Protocol

Chord layers on different nodes communicate with the Chord protocol (defined in assignment sheet)

Protocol is text-based

- Protocol has absolutely nothing to do with other implementations of Chord

Joining: JOIN, JOIN_OK, NEWNODE

Leaving: LEAVE, NODEGONE

Store: STORE

Retrieve: RETRIEVE, OK, NOTFOUND

Misc: TRANSFER (join + leave)

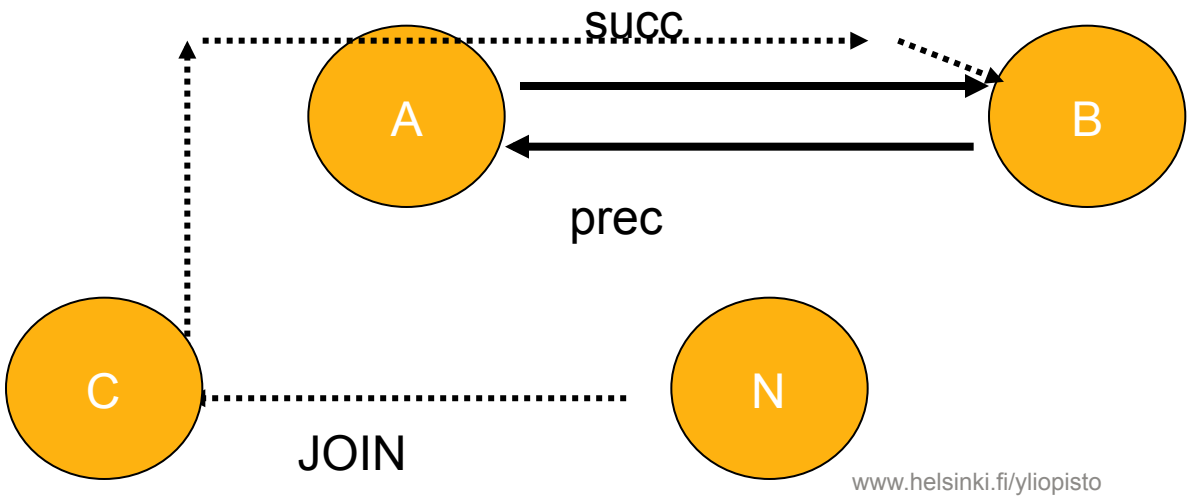
Joining the Chord Ring

Need IP address and port of an existing node!

Node picks its own ID, ID = hash(IP:port)

Protocol messages:

- JOIN: New node sends to known node (C), forwarded to node responsible for new node's ID (B)
- JOIN_OK: Reply to new node (+ TRANSFER)
- NEWNODE: New node sends to its predecessor

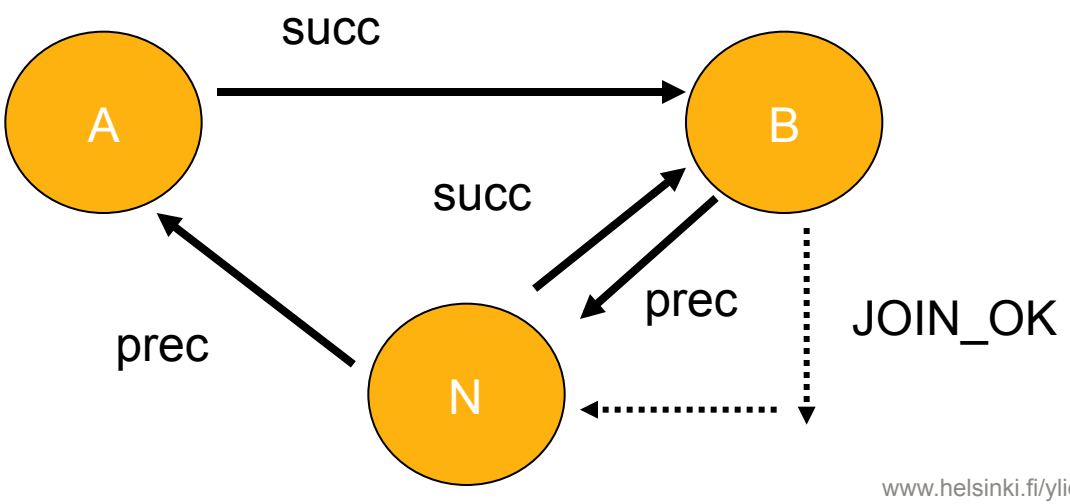


Joining the Chord Ring

Need IP address and port of an existing node!

Protocol messages:

- JOIN: New node sends to node responsible for its ID
- JOIN_OK: Reply to new node (+ TRANSFER)
- NEWNODE: New node sends to its predecessor

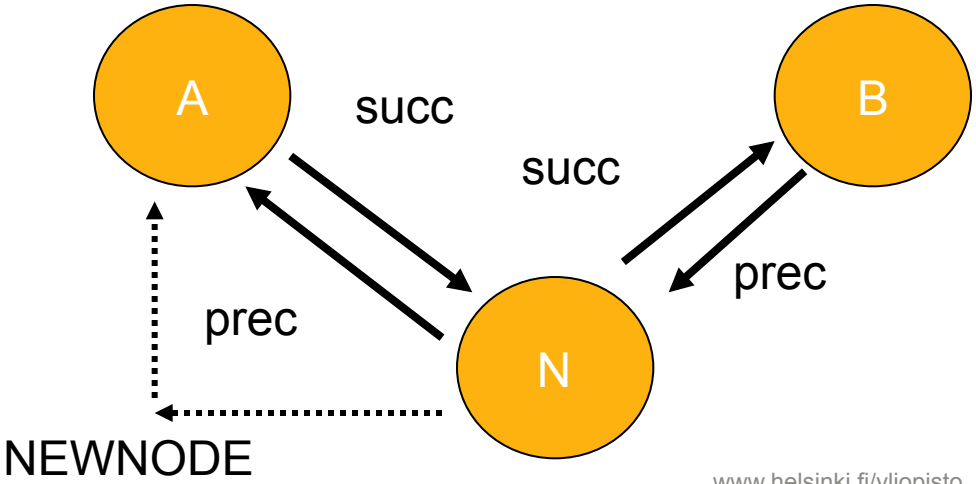


Joining the Chord Ring

Need IP address and port of an existing node!

Protocol messages:

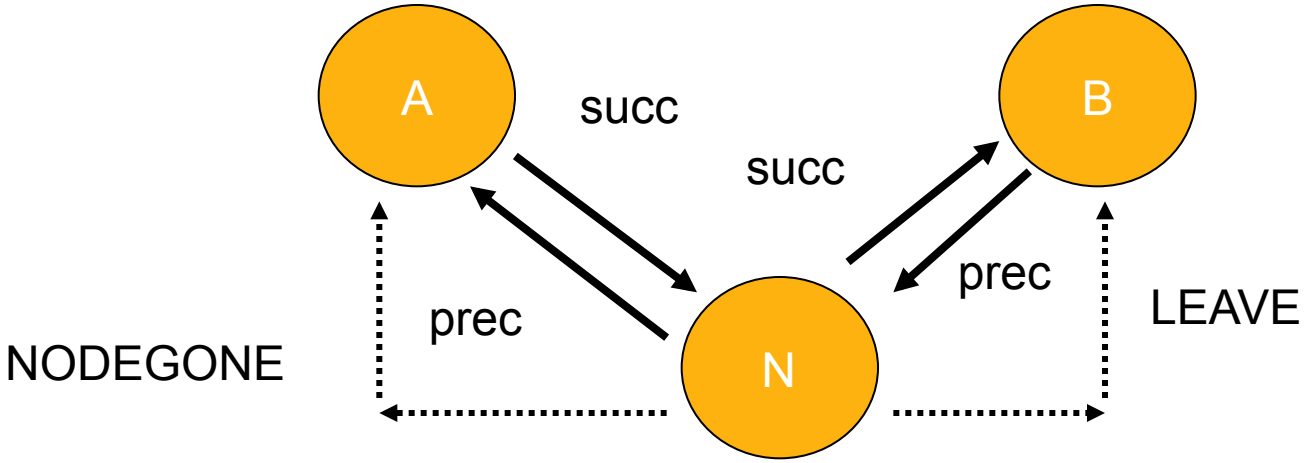
- JOIN: New node sends to node responsible for its ID
- JOIN_OK: Reply to new node (+ TRANSFER)
- NEWNODE: New node sends to its predecessor



Leaving the Ring

Messages:

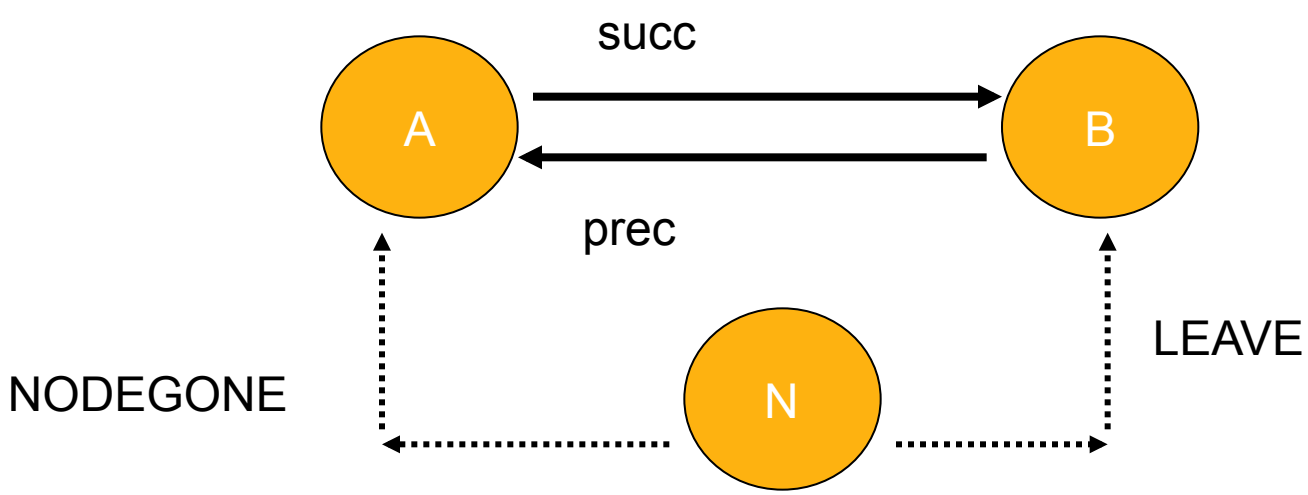
- LEAVE to successor (+ TRANSFER)
- NODEGONE to predecessor



Leaving the Ring

Messages:

- LEAVE to successor (+ TRANSFER)
- NODEGONE to predecessor



Storing and Retrieving

STORE and RETRIEVE (and OK + NOTFOUND) passed always along the ring

Forward for STORE + RETRIEVE, backward for others

To store object with name “Foo” and value = 5

1. Calculate id = hash(“Foo”)
2. Send message STORE id 5<CRLF>
3. Message routed to node responsible for id

Each node must store key-value pairs for which it is responsible

How you organize the storage in a node is your business

But: Need for retrieving single objects and ranges

How to Process Messages

Messages handled recursively

- Node A sends to node B, node B to node C, ...
- Reply (if any) comes back on the reverse path

Messages to be forwarded on ring

- Forward: JOIN, STORE, RETRIEVE
- Backward: JOIN_OK, OK, NOTFOUND

All others always between two nodes directly

Use TCP, keep connections open for replies

- Replies for JOIN, RETRIEVE

No need to keep permanent connections to predecessor and successor

Hashing / Joining the Network

Simple hash functions sufficient

Must implement some way of specifying hashes manually at run-time!

- Helps (a lot) with debugging and testing
- Ok to modify API for this feature

Joining needs address of an existing node

How about first node?

- Special case

Traps and Pitfalls

Responsibility ranges of nodes

- Node is responsible for all keys between its predecessor and itself (its own ID included)
- Think (a bit) before programming!

Keeping track of predecessor and successor

- Absolutely vital for correctness of network
- Think (a lot) before programming!

How to organize code in Chord layer

- Very easy to write spaghetti code
- Think before programming!

Miscellaneous

Use many nodes when testing (can be on one computer)

- First make sure 2 nodes work
- Then really make sure 3 nodes work
- Then check with 4 and prove to yourself that it works
- Use at least 4 nodes once your code works

Compatibility

- In principle, everything should be compatible
  - Application from A, Chord layer from B, other Chord nodes from other students, ...
- In practice, we do not require compatibility
- Extra points for compatible implementations! ☺

Milestones

Suggested order for assignment

Milestone 1

- Small application

Milestone 2

- Basic Chord functionality, join, leave, transfer (empty)
- Check carefully that this works properly!

Milestone 3

- Store and retrieve values

Milestone 2 is most important

- Heaviest weight in grading of assignment and required for passing

Things NOT Needing to be Implemented

Fault tolerance

- If one node crashes, the whole network can crash

Security features against malicious nodes

- If a node is evil, the whole network can crash

Finger tables

- Difficult to get right
- Test networks too small to see any benefit in any case