
# LABs on DISTRIBUTED HASH TABLES

## Implement a minimal DHT in pure Python

Lorenzo Ghiro

*lorenzo.ghiro@unitn.it*

# Outline / Goals

UNIVERSITY OF TRENTO - Italy
Information Engineering
and Computer Science Department

- Academic exercise: Make practice with DHTs implementing a little one in pure Python
  - ➢ step-by-step exercise
- Spoiler #1: 100 lines of code are almost enough :)
- Spoiler #2: The DHT implementation is really basic, next lab we sophisticate it and make it more realistic

# Introduction



- A large number of research DHTs have been developed by universities and companies
- Each DHT scheme generally is pitched as being an entity unto itself
  - The various available schemes are actually all pretty similar: Take one, make a few tweaks, and you end up with one of the other ones
- In order to fully express the spectrum of options, let's start with a basic design and then add complexity in order to gain useful properties

# DHT in a nutshell



- a DHT performs the functions of a hash table
- You can store a key and value pair, and you can look up a value if you have the key
- The interesting thing is that storage and lookups are distributed among multiple machines
- Unlike existing master/slave database replication architectures, all nodes are peers that can join and leave the network freely

# Towards python coding...



- Sketch idea: we start with a **circular, double-linked list**
  - Each node in the list represents a machine on the network.
- Each node keeps a reference to **successor** & **predecessor** nodes in the circular space
- We must define an **ordering** so we can determine who are successor and predecessor for each node of our DHT

# Address Space



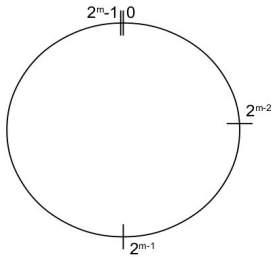
- For example, Chord determines the successor node this way:
  - Assign a unique random ID of k bits to each node
  - Arrange nodes in a ring, with IDs in increasing order clockwise around the ring
  - For each node, the successor is the node with the smallest clockwise distance
  - For most nodes, this is the node whose ID is closest to but still greater than the current node's ID
  - The one exception is the node with the greatest ID, whose successor is the node with the smallest ID

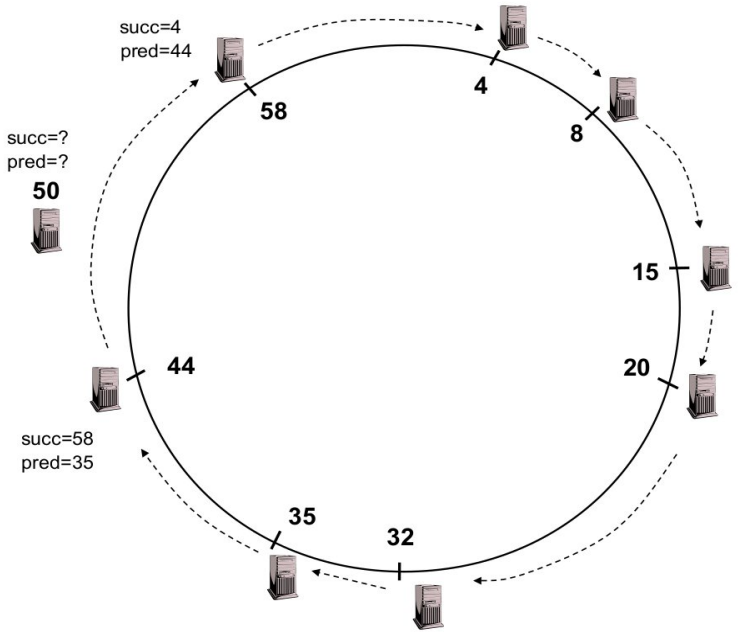
# Address Space



## Chord

- ID space: uni-dimensional ring in $[0, 2^m - 1]$ $(m = 160)$
- Routing table size: $O(\log n)$
- Routing time: $O(\log n)$



# First steps: hash and distance methods



- STEP 1: define in python an **hash-based function** that, for a given input, **generates a digest of k bits**
  - this function shall be used to compute nodeIDs and thus to place them on a modulo 2**k - 1 ring
- STEP 2: define a **"clockwise_distance" method** that, given two numbers X and Y (unsigned int of k bits), returns the number of "ticks" that a clock starting on X should do to reach Y

# Coding Time!

# Before implementing JOIN



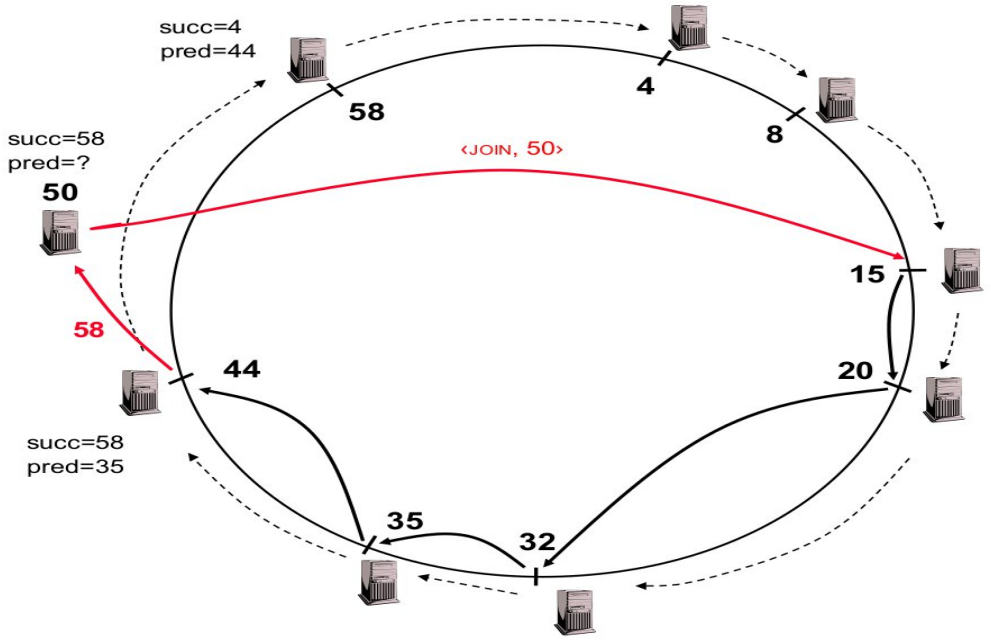
- STEP 1: define a **NodeDHT class**
  - name, ID (address), pointers to successor and pred, storage (an HT)
- STEP 2: define a **main()** where you **setup a network**
  - with two initial nodes (n1, n2), constituting our initial linked list
  - n1.succ --> n2  n2.succ --> n1
  - n1.pred --> n2  n2.pred --> n1
- SPOILER: Next step will be the implementation of JOIN mechanism to add more nodes in our DHT

# Coding Time!

# JOIN illustration



- Be X the new comer…
  A --> X --> B  or  B --> X --> A ???
- if you have a good clockwise_distance it's easy to understand the criterion for choosing where to place newcomers

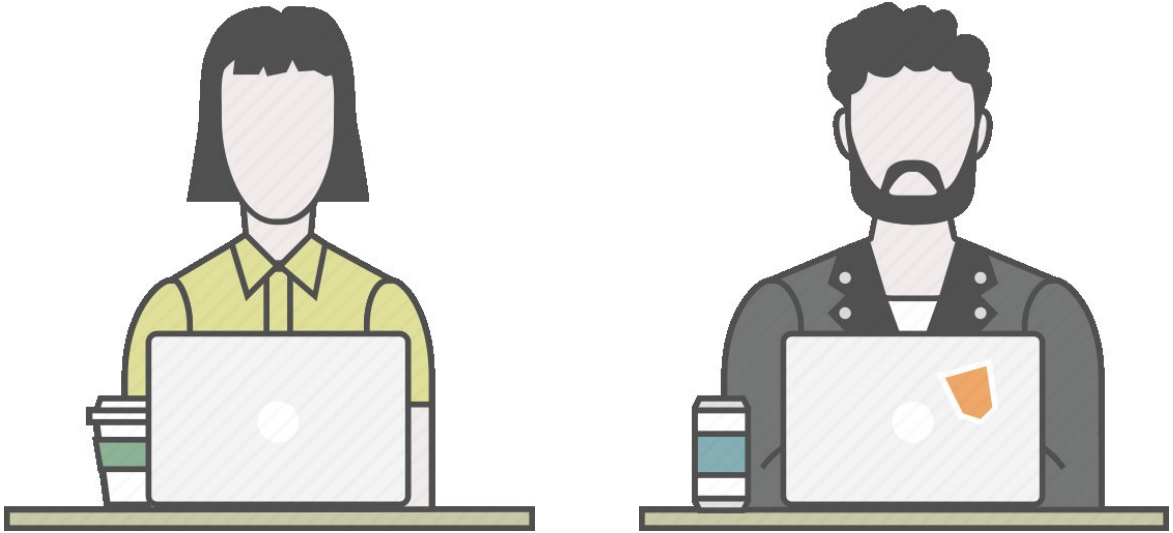


# JOIN criterion



- When a node N receives a JOIN request from a requester R
- if `dist(N, R) > dist(N.succ, R)`, then it means R should not JOIN in between N and N.succ
  - ➢ RECURSIVELY forward the request to N.succ!
- ...else, i.e., `dist(N, R) <= dist(N.succ, R)`
  - then link properly R between N and N.succ
- TEST: in main() create 2 further nodes (n3, n4) and let n3 join the DHT from n2, while n4 join from n1

# Coding Time!



# STORE and LOOKUP



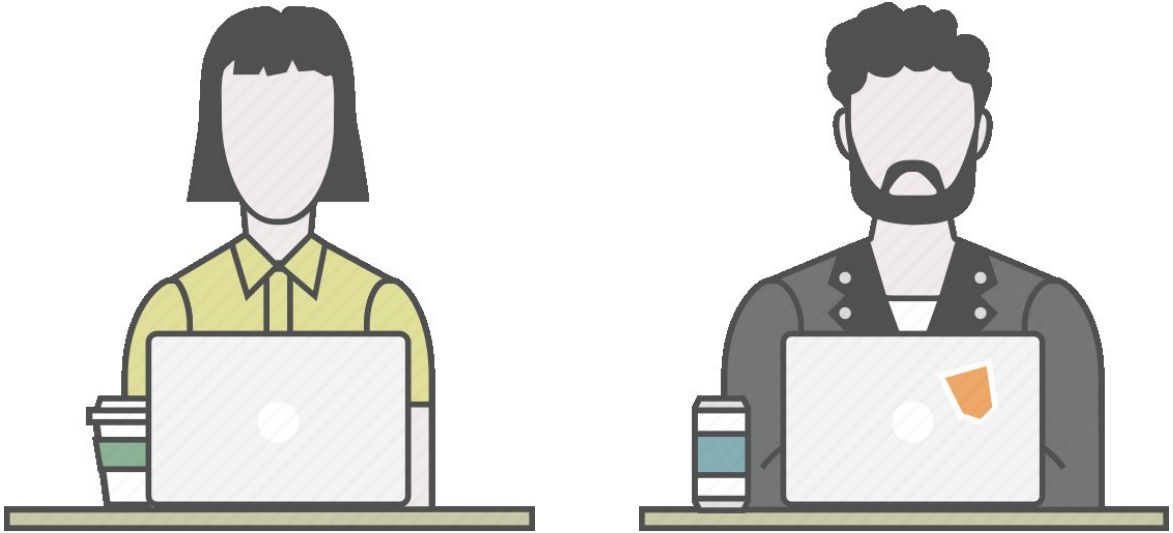
- To store or retrieve an item from the DHT you need to find the node responsible for the related key
  - then do a normal hash table store or lookup in that node
- How to choose the node responsible for a given item?
  1. Take the item and hash it, generating a key of exactly k bits
  2. Treat this key as a node ID ⇨ determine which node is its successor
  3. as for JOIN: start at any point in the ring ⇨ go clockwise until a node is found whose ID is closest to but still greater than the key
  4. DONE! The found node should store the given item

# Implement STORE and LOOKUP

UNIVERSITY OF TRENTO - Italy
Information Engineering
and Computer Science Department

- Define `store(key, value)` and `lookup(key)` as methods of the **NodeDHT** class
  - Forward request recursively if needed
  - If key not available in the DHT, the return None
- TEST: Populate DHT issuing many `store` with random content assigned to random nodes
- TEST: Issue some lookups, also of not-inserted content
  - nice printing to see chain of forwarding :)
  - Items are fairly assigned to the various nodes???

# Coding Time!



# Questions?