



Fall 2009


Dr. Kevin Hamlen



# CS 4485: CS PROJECT



# The Project

- build an advanced P2P file-sharing client
  - main features
    - keyword search
    - P2P file download
    - graphical user interface
    - automated test suite
    - user manual, technical manual
  - project organization
    - teams of 3-4 members each
    - you choose the programming language(s), development methodology, etc.
    - the customer (me) chooses the specs & deadlines
- 

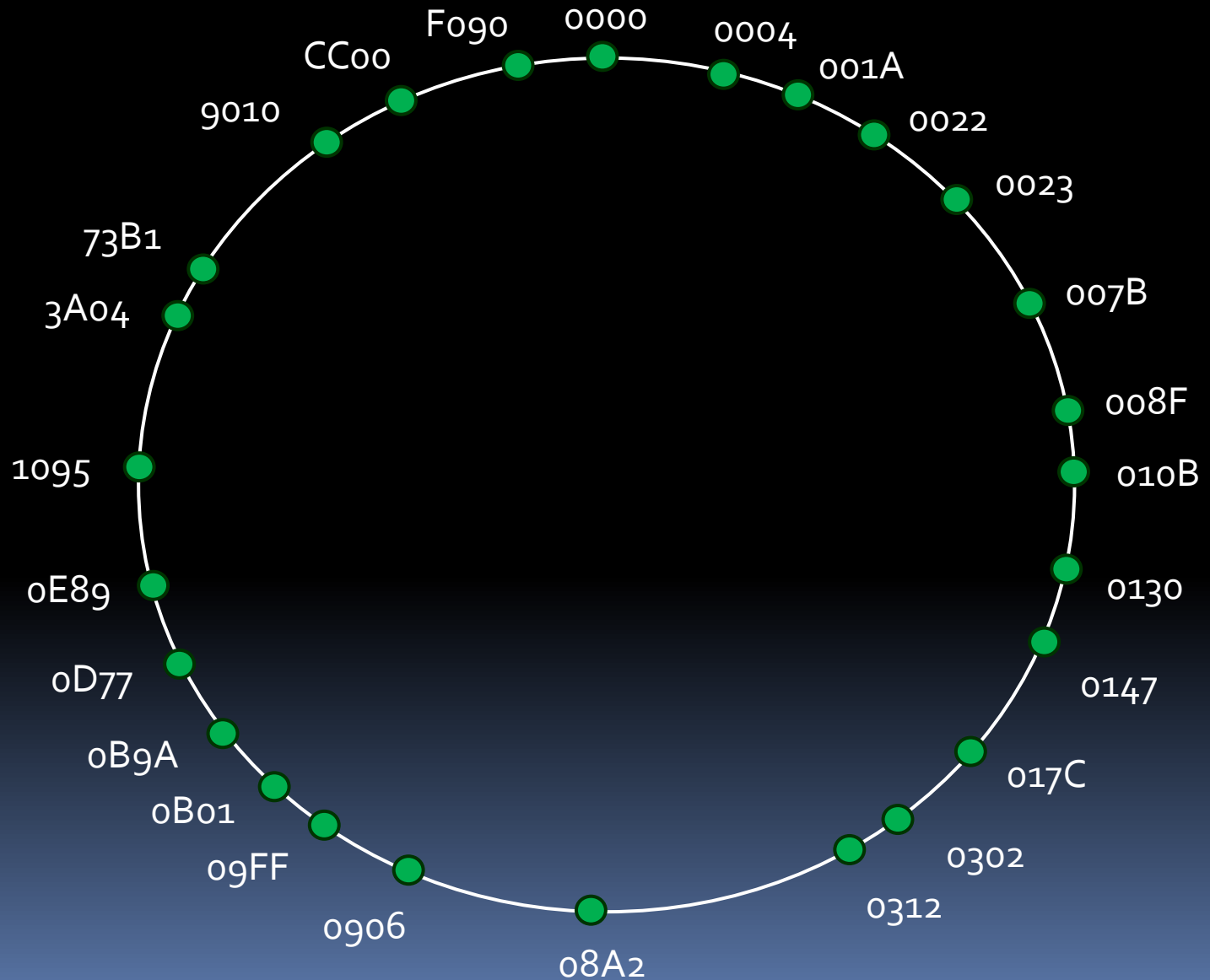
# By 5:00pm today...

- You should be a member of a group of 3-4
- Your group should have emailed me a group description and work plan
  - I will assign you a grade of 1/1 for the Work Plan assignment in eLearning once you've satisfied this requirement
- Stage 1 is due in three weeks (9/18)

# Chord Overlay Structure

- Every client gets a unique number
  - called the client's "identifier"
  - computed by applying a **secure hash function** to the client's IP number & port number
  - Example:  $H("127.0.0.1:10035") = \text{0xF382E26A}$
- secure hash functions
  - map values from a large domain (e.g., strings) to a smaller domain (e.g., N-bit integers)
  - one-way functions (hard to reverse)
  - collisions very rare (astronomically improbable)
  - NEVER TRY TO INVENT YOUR OWN!
  - choose a good hash function (do some research)
  - select a good N based on your choice (e.g., N=160)

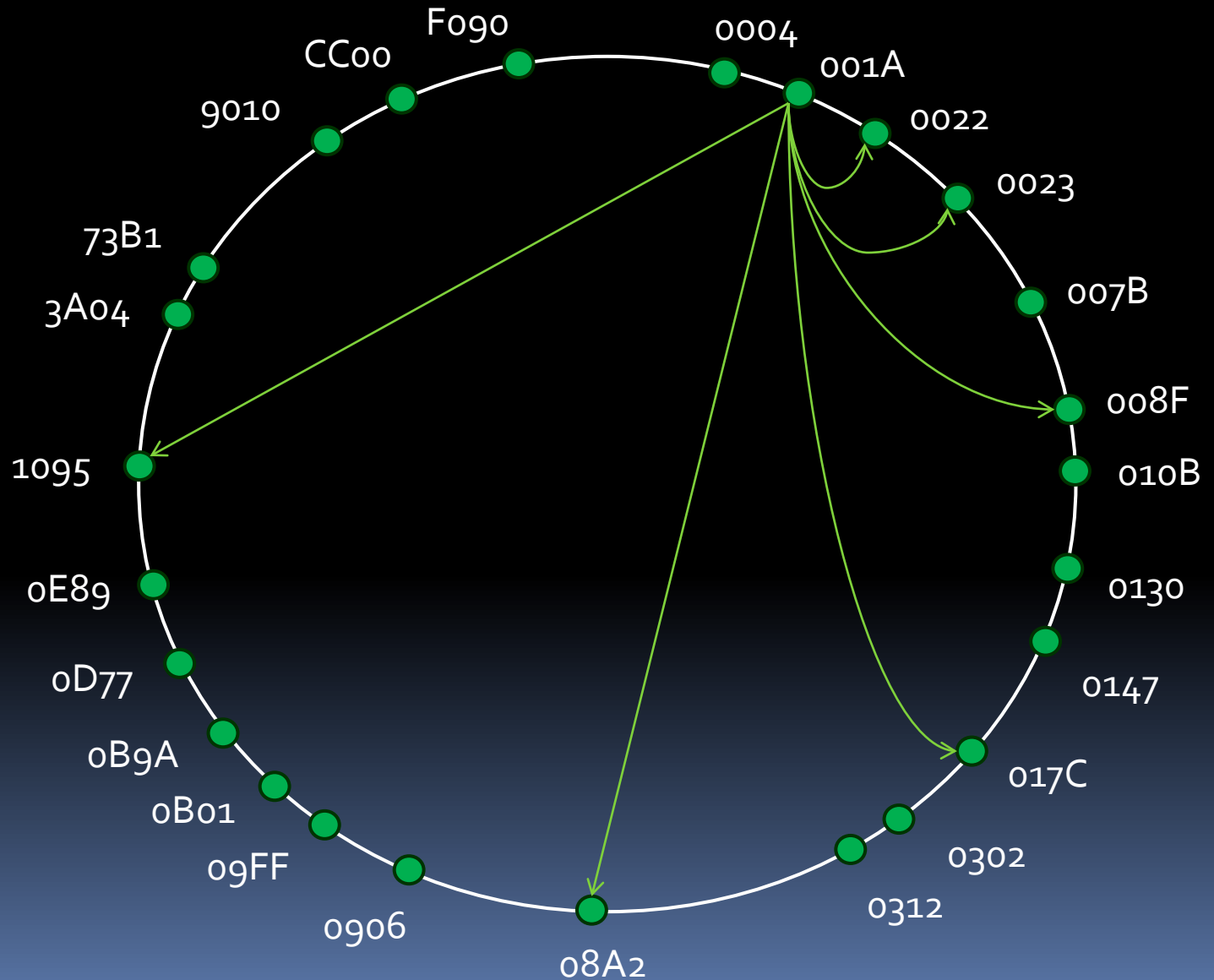
# Identifiers form a Ring



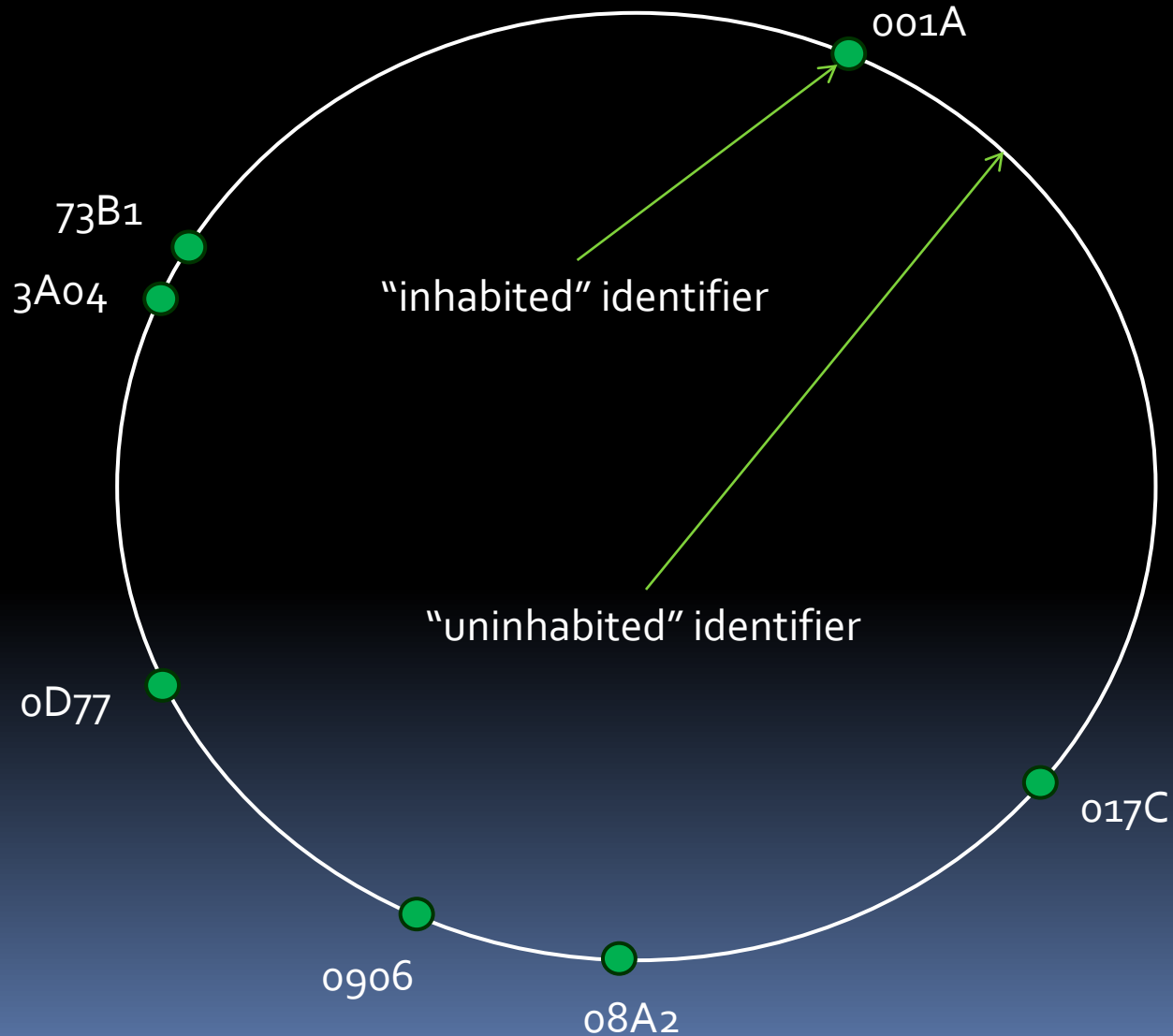
# Chord Finger Tables

- Each peer keeps track of a table of  $N$  other peers
  - Let  $d$  be my identifier
  - The  $n$ th entry in my table ( $0 \leq n < N$ ) is the IP and port number of the peer whose identifier is nearest to  $(d + 2^n) \bmod 2^N$
  - So the first entry is the next peer in the ring (my successor)
  - Each peer in the table is about twice as far away from me as the previous one in the table
  - Special case: If no peer has an identifier less than  $(d + 2^n) \bmod 2^N$ , then use the peer with greatest identifier

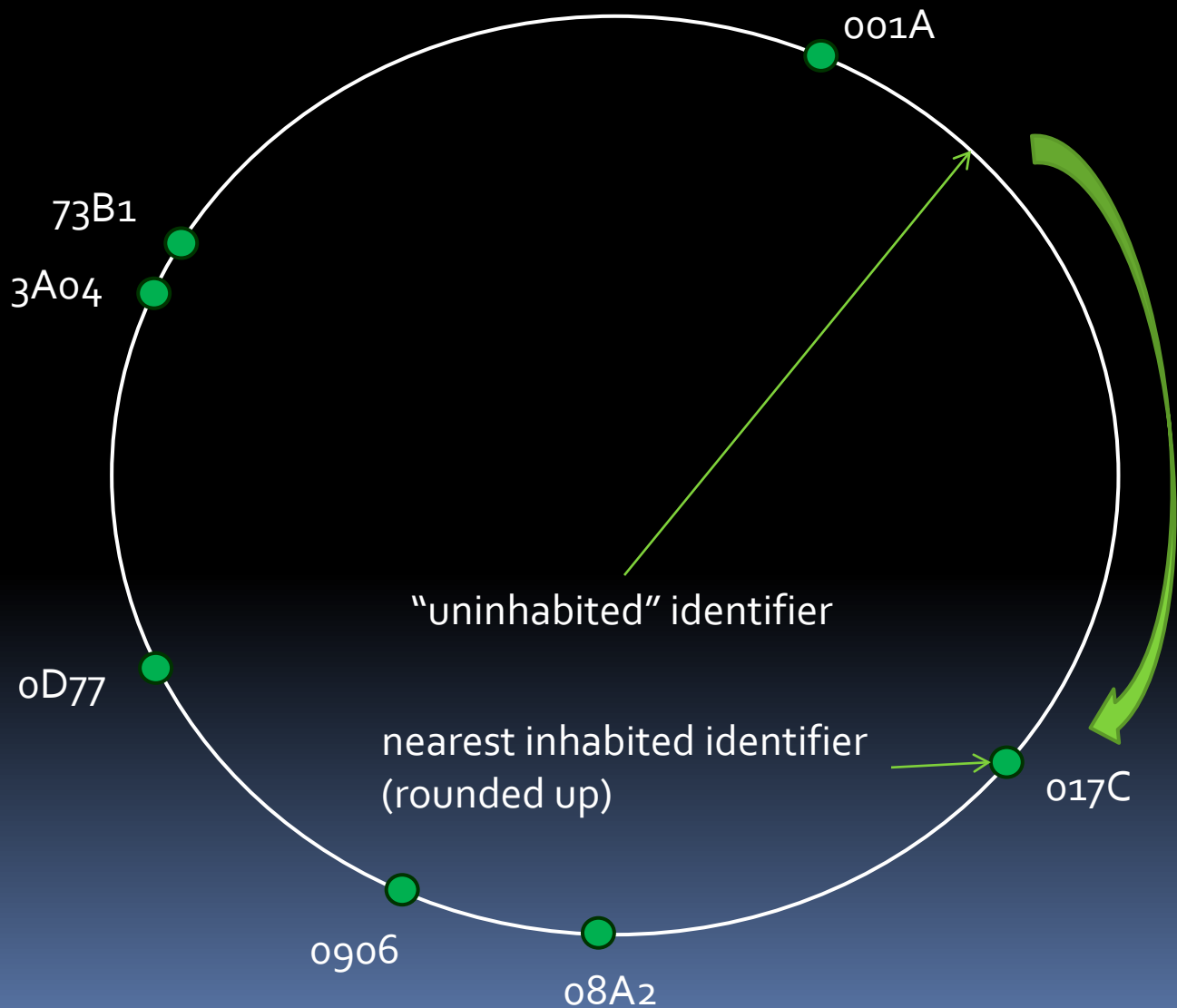
# Example



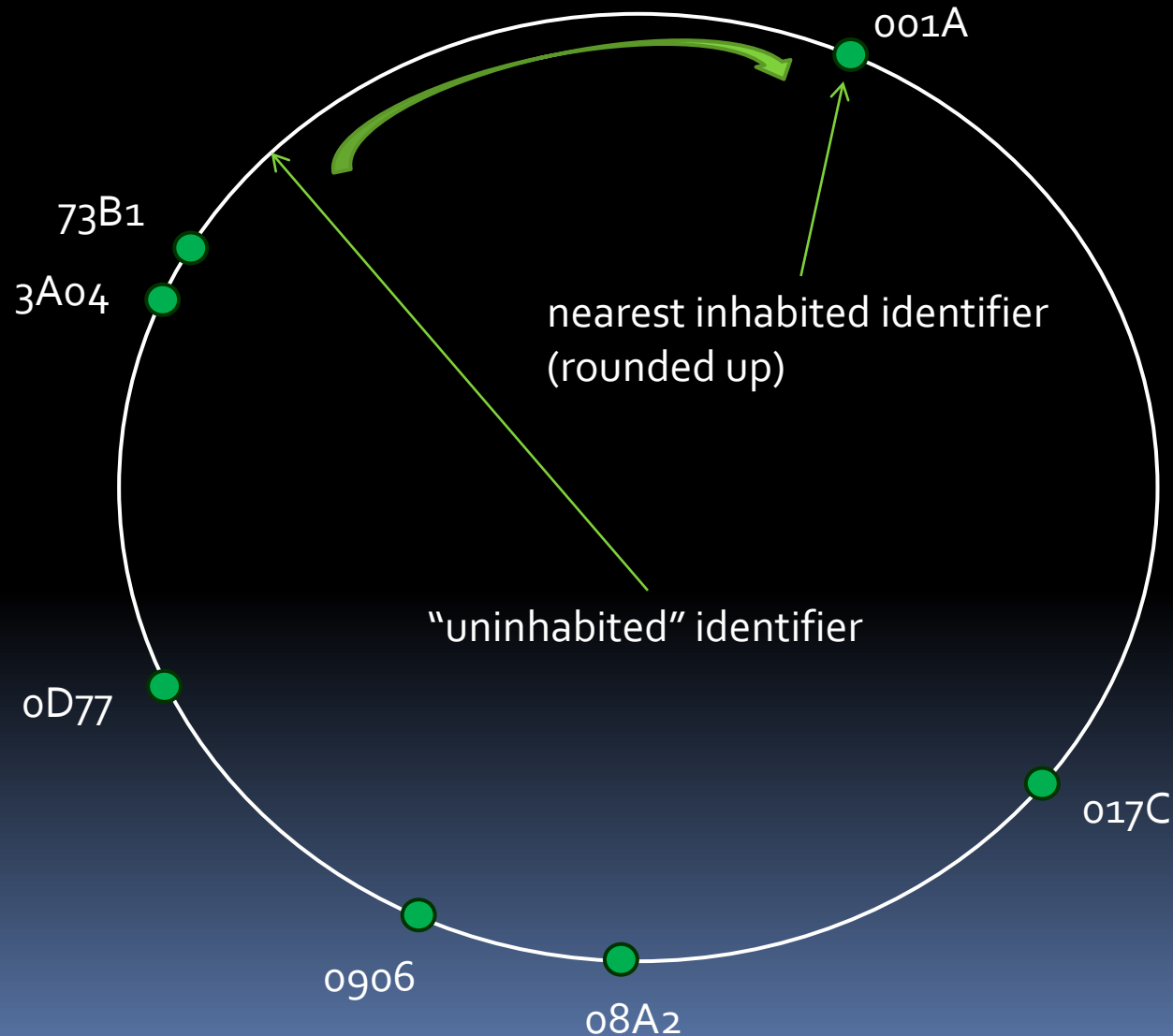
# Some Convenient Terminology..



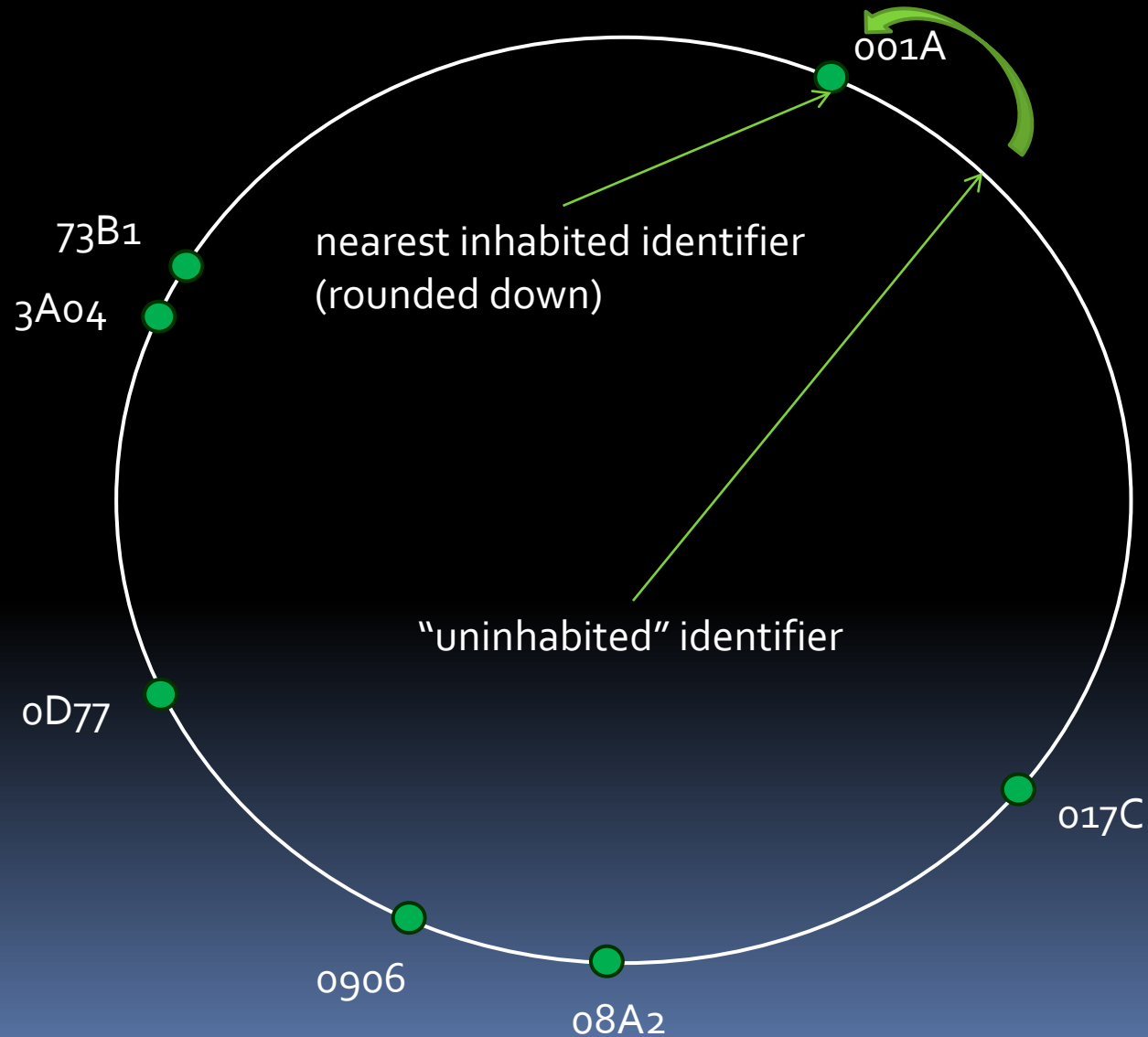
# “Round up” (clockwise) to the next inhabited identifier



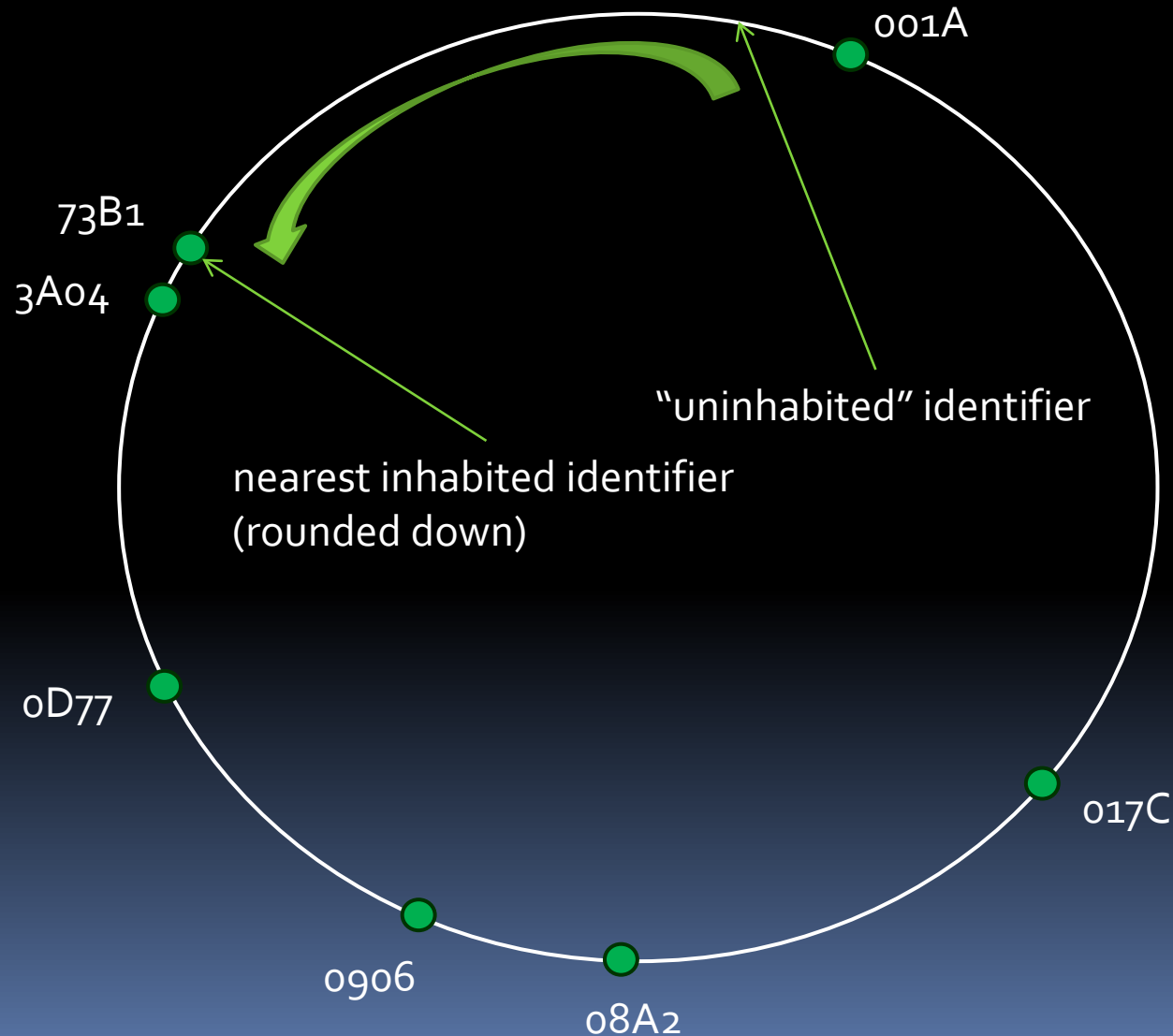
Note: “Rounding up” can result in a LOWER identifier!



# “Round down” (counter-clockwise)



Note: "Rounding down" can result in a GREATER identifier!



# Chord Finger Tables

- Each peer keeps track of a table of  $N$  other peers
  - Let  $d$  be my identifier
  - The  $n$ th entry in my table ( $0 \leq n < N$ ) is the IP and port number of the peer whose identifier is nearest to  $(d + 2^n) \bmod 2^N$ , **ROUNDED UP**
- Observations
  - The first entry is always my successor
  - The same peer might appear more than once in my finger table
  - I might appear in my own finger table

# Message Send/Receive

- To send a message to the peer whose identifier is nearest  $x$ , **ROUNDED DOWN**:
  - If  $x$  is “between” me and my successor, don’t send!
    - Recall that “between” has a special case for when my successor’s identifier is not greater than mine.
  - Choose the peer in my finger table whose identifier is nearest  $x$ , rounded down.
    - What if that peer is me?
  - Send the message to that peer and ask him to forward it to  $x$ .
  - Process continues until the desired peer is reached.
- Guaranteed to take no more than  $N$  hops

# Stage 1 Objectives (due 9/18)

- Implement a basic P2P overlay in your language of choice
  - feel free to use publicly available, standard libraries (must be used in accordance with relevant licenses, of course)
  - existing P2P libraries, tools, etc., are off limits (they are your competitors)
- Required functionality
  - list of all peers fixed at start time (no join/leave)
  - each peer can send a message (e.g., a string) to the peer whose identifier is closest to  $x$ , rounded DOWN
- Two interfaces supported
  - a graphical user interface (human user)
  - an API (to use the client as a service)
- Implement a test suite
  - uses the API to make clients on the same machine (listening on different ports) send messages
  - tests to see if messages were correctly received and no spurious messages received

# Initialization

- Client takes its own port number as an input parameter
- Client starts by reading a configuration (text) file that lists the IP number and port number of ALL clients in the network (including itself).
  - Choose a nice syntax for the config file (e.g., XML?)
- Client computes its finger table using that list, then discards the list.
  - Don't use the list to cheat and just send messages directly to recipients! (Future stages will not work if you do.)

# Graphical Interface

- Stage 1 interface should have three widgets:
  - Message box: Lets me type a string to send to another client
  - ID box: Lets me specify an identifier to which the message should be sent
  - SEND button: Sends the message (using the Chord protocol) to the peer whose id is nearest  $x$ , rounded down.
- Interface shows two forms of activity:
  - Message received (show text)
  - Message forwarded (from whom to whom)
- Interface should be multithreaded!
  - GUI's should always be responsive to the user no matter what else might be going on.
  - Example: No GUI freezes while sending a message.

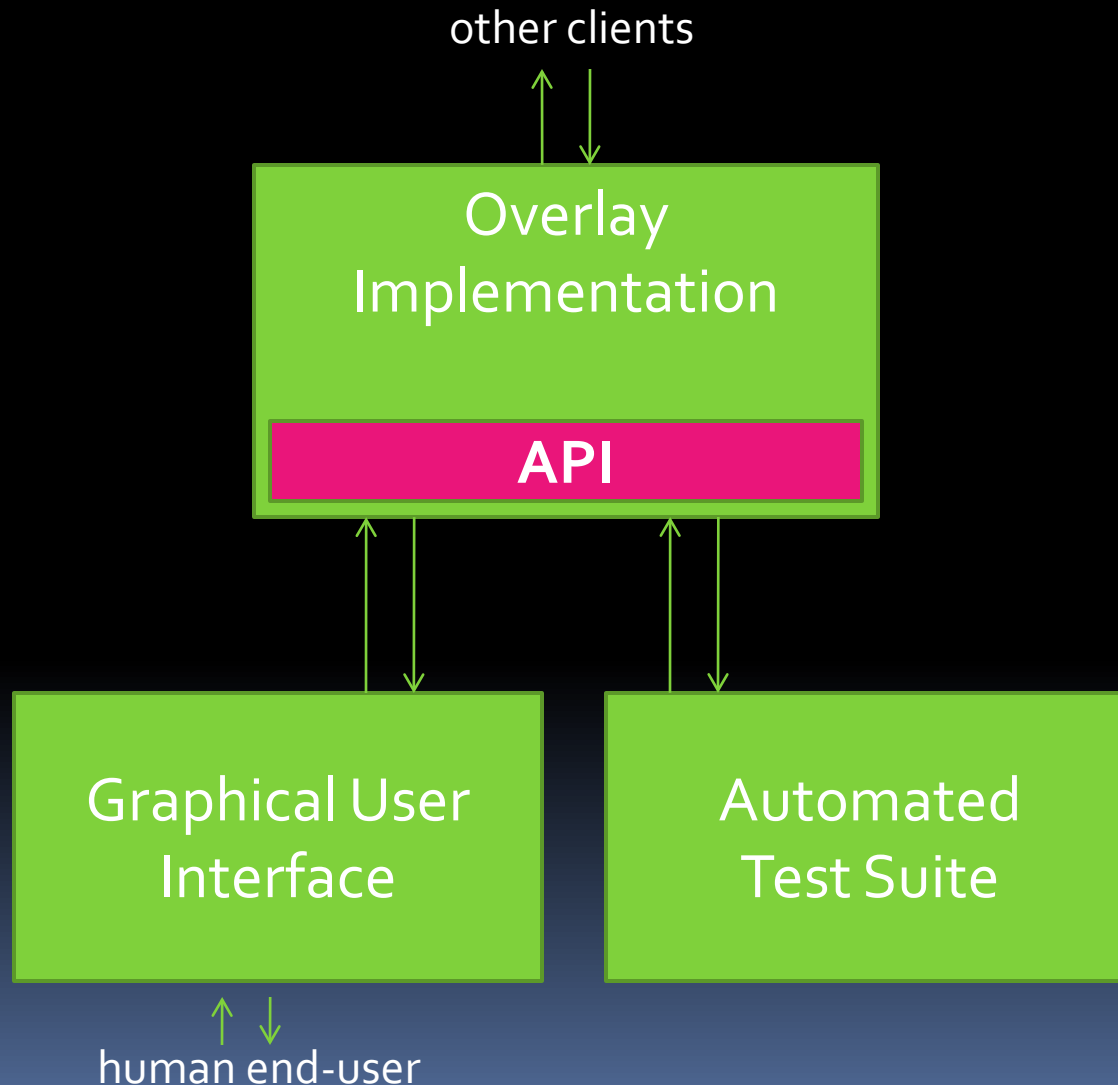
# Automated Testing

- Test suite drives MULTIPLE clients on the same machine
  - (the clients listen on different ports)
- It instructs clients to send messages
  - pseudo-random (but choose the seed value deterministically so test can be repeated!)
  - messages sent at random times (don't wait for each to be delivered to send the next)
  - test that each message was received by the correct client, and no spurious messages received
- You choose the exact testing regimen, however:
  - pay special attention to corner cases (e.g., messages that "wrap around" the Chord ring)
  - test failures must be reported AUTOMATICALLY and CLEARLY (no data-mining for errors, please)

# Automated Testing

- Possible approaches
  - Implement overlay as a class library and have the test suite invoke multiple instances of the class
  - Implement test suite as a script that launches each client as a separate child process and communicates with each via channels (e.g., stdin/stdout)
- GUI may be open as test suite runs, or not
- Test suite can have any sort of interface
  - command-line, text file, etc.

# High-level Structure



# Stage 1 demo (in class 9/18)

- You demonstrate your client for me for 20 min
  - launch some clients (e.g., 10) on one machine
    - bring a laptop or take me to a lab
  - send a few messages using the GUI
  - run a few automated tests and explain how they work
  - make optimal use of your limited time! (i.e., be ready to go)
- Written report (2-5 pages)
  - describes current state of your project and the contributions of each team member
  - describes testing methodology and argues for comprehensiveness of tests
  - lists any bugs as yet undiagnosed

# Time Management

- Main concern should be implementation
  - less diagrams, more code
- Try to have implementation “done” in two weeks
  - at that point you will discover a week’s worth of bugs to occupy you until the due date
- Some bugs are to be expected at this early stage, but not so severe that they prohibit the basic functionality required for the demo!
- Report can be started and finished the day before the due date

# If you can't make the deadline...

- Demo whatever you have completed
- Your Stage 1 grade will suffer but it's only 10%, so it's not a catastrophe
- But you will have a harder time completing Stage 2 on time, so don't let it snowball into a catastrophe!



Questions?