Structured P2P Networks (Chord and DHT) (10 points)

This homework is focused to establish a structured P2P overlay network (Chord). Please investigate the theoretical background of Chord networks in details https://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

The solution for this homework must be uploaded to OLAT until Monday, **07.05.2018**, at **08:00**, and will be evaluated on 9th of May.

Assignments:

1. Routing in Chord

   Within a Chord network, objects (peers and data) are mapped to a linear address space spanning the interval [0, 2^m - 1], where *m* is some constant (typically 128 or 160). The mapping is realized via a hashing function (e.g. SHA-1). By considering the object with the address 0 to be adjacent with the object 2^m - 1, a ring is formed (Chord Ring). For this example, consider *m* = 4. Hence, our network can contain up to 16 different objects.

   a. Draw the Chord ring and place nodes on maximum five addresses calculated with the following algorithm applied to your Matrikelnr *abcdefg*:

      a+b+c+d+e+f+g mod 16,
      b+c+d+e+f+g mod 16,
      c+d+e+f+g mod 16,
      d+e+f+g mod 16,
      e+f+g mod 16,
      f+g mod 16 and
      g mod 16.
      In case some digit is 0, consider it as 10. If your calculation results in less than five addresses, use only those. **(0.5 points)**

   b. Compute the Finger table of the nodes with the node with minimum (MIN) to node with maximum (MAX) addresses that you calculated and illustrate those within your drawing. **(0.5 points)**

   c. Peer MIN wants to send a message to the peer MAX. Illustrate the necessary steps, assuming iterative and recursive ways. **(1 point)**

   d. Peer MAX wants to send a message to the peer MIN. Illustrate the necessary steps, assuming iterative and recursive ways. **(1 point)**

2. Adding / Removing Nodes to/from a Chord Ring

   In addition to the possibility of supporting an efficient routing mechanism between pairs of nodes within the network, P2P systems also have to be capable of supporting efficient operations for joining and leaving nodes. Apart of the finger tables, every node also maintains a reference to its immediate successor (first finger table entry) and predecessor on the chord ring. This simplifies the join / leave algorithms.

Joining the network:
(one node at a time only – simplified version of the paper's algorithm) Let ***n*** be the new node (***n*** is its address). When joining the network, the new node has to: *i*) find its location within the ring, *ii*) fill its finger table and finally *iii*) update the finger tables of other nodes which should now point to the new node.

i. To join the network, the new node needs to know one node that is already connected. The new node is using this node to conduct a lookup operation, searching for the node succeeding the location ***n***, hence the location the new node should be added to. The new node is now integrated into the ring by manipulating the predecessor / successor references of the located node and its successor (as for linked lists)
ii. To fill its own finger table, the new node simply queries for the "ideal" addresses the finger pointer should be pointing to ($n + 2^{i-1}$) and stores the resulting node addresses.
iii. To update the finger pointers of other nodes, the corresponding nodes have to be located first. The following pseudo code fragment does the trick (remember, *m* is the number of finger table entries)

```
for (i=1 to m) {
    // find last node p whose ith finger might be n
    p = lookup_predecessor(n-2^(i-1));
    // update fingers counter-clock wise as long as necessary
    while (p.getFinger(i) > n) {
        // update ith finger
        p.setFinger(i, n);
        p = p.getPredecessor();
    }
}
```

Leaving the Network:

In Chord, no active modifications are necessary when leaving the network. This property is especially useful when facing sporadic failures of nodes. However, to ensure the validity of the routing information, every node within the network has to verify/update its finger table entries periodically (just like during the join operation). Further on, the successor and predecessor links have to be kept valid. In case some node loses its successor, it can ask the node within the 2nd entry of the finger table for its predecessor and iterate back until it finds its new predecessor. When updating the successor reference, also the successor's predecessor is updated. This way, lost predecessors are recovered as well.

a. Demonstrate a node joining the network as given within 1a) at a random position you choose. **(3 point)**

b. Based on the state of 1b), assume the successor of the node you added is failing. Demonstrate the modifications done within the finger tables. **(4 point)**