# **Project 3 ReadMe**

## Team members:

Isabel Laurenceau 7393-5064

Anshika Saxena 9530-5566

Input: To run project three use ‘mix run project3.exs [number of nodes] [number of requests]’

Output: Maximum number of hops traversed for all requests for all nodes.

## What is working:

The project3.exs reads the input arguments and sends them to the MAINPROJ module. The MAINPROJ starts the dynamic supervisor TAPESTRY module, starts all the nodes from TAPNODE module and adds them to the Tapestry mesh. Once they are all inserted to the mesh the MAINPROJ tells each node [number of requests] random objects from the Tapestry mesh.

#### JOIN

The TAPNODE initializes all nodes with a randomly created number which is hashed using the SHA1 algorithm to produce it’s id. It is also initialized with an empty neighborMap, empty objectList and empty objectLinksList. The id and three lists will be used later for inserting into the tapestry and routing.

To add each node to the tapestry *addToTapestry*is called. Here we contact a contact Gateway Node using *contactGatewayNode* which returns the id of a node already in the Tapestry mesh. We (node N) than use this gateway node to route to where we should be in the mesh and fill in our neighborMap. *hNodeToRoute* first sends a hello message to nodes it encounters so that they made add us (node N) to their neighborMap.

When a node (H) receives a ‘Hello’ message from node N it uses *placeInNeighborMap(state, neighbor\_id)* to add it to it’s neighborMap. *placeInNeighborMap* first finds the longest matching prefix between node H and node N’s ids. It than finds i, which is the element in the index of the next id digit after the prefix of N. It uses j and i to place N in its neighbor map. If there is already an element in that location it uses *updateYourNeighborMap(j, my\_neighborMap, new\_neighbor)* to select the primary and backup links. If not it adds N to its neighbor map and tells B to add it to its neighbor map since its bidirectional.

For N to populate its neighbor map it uses routing. If there is an element where it would be in B’s routing table it routes to that elements and tries to find a possible closer neighbor. It continues this until no neighbors are available. An interesting observation made was that as the network is small it’s more likely you will not have anything in common with the gateway node and that the gateway node will not have anything in common with its neighbors. This leads to very large first levels and an almost fully connected network. This is not the case with larger networks as it is more likely that you “match” prefixes with other elements and can better place yourself.

#### ROUTING

To Route an object MAINPROJ tells a node which id to route towards using *sendRequest*.

## What is the largest network you managed to deal with:

## Program- High Level :

**Script**

R reads arguments and sends them to the MAINPROJ module.

**MAINPROJ**

Start Supervisor

Create number of nodes

Create overlay network

Send requests from every node

**TAPESTRY**

Init Supervisor

**TAPNODE**

State

Get created

addToTapestry()

contactGatewayNode()

updateYourNeighborMap()

sendHello(new\_neighbor, N)

receiveHello(new\_neighbor, N)

placeInNeighborMap()

sendNeighborMap(new\_neighbor, N)

optimizeNeighborMap()

lookupNeighborMap()

notifyNeighbors(surrogate(new\_id))

notifyEmpties()

sendFirst(childPid, *numRequestToSend* )

nextHop(n, G)

routeToObject (OG)

publishObject(OG)

unpublishObject(OG)

routeNode(N, Exact)

## Program- Pseudocode:

**Main**

Read input arguments{

Take command line arguments

Make them into integers

Pass the integers to MAINPROJ

}

**MAINPROJ**

Start Supervisor {

{:ok, \_pid} = Tapestry.start\_link(1)

}

Create number of nodes{

rng = Range.new(1, numNodes)

for x <- rng do

Tapestry.start\_child(x, numRequests, neighbor\_map)

end

}

Create overlay network{

for every child in Tapestry.whichchildren do

TAPNODE. addToTapestry (childPid)

end

}

Send first request from every node{

for every child in Tapestry.whichchildren do

TAPNODE.sendFirst(childPid, *numRequestToSend* )

end

}

**TAPESTRY**

Init Supervisor

**TAPNODE**

State**{**

*new\_id*

*numRequestToSend*

neighborMap - Routing Table

**}**

Pseudocode from Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing

**Get created {**

def init(*numRequestToSend*){

* Node *N* requests a new ID *new\_id*
  + *new\_id* = SHA-1(….?)
* *numRequestToSend = numRequestToSend*
* neighborMap = empty list
* ROUTER = ??
* DYNAMIC\_NODE\_MANAGEMENT == ?

}

def start\_link(*numRequestToSend*){}

}

addToTapestry{

* Call contactGatewayNode(); returns G
* H = G;
* For (i=0; H != NULL; i++) {//terminate when null entry found
  + Send Hello to neighbor no matter what so they can check if they need to add me to their map
  + Grab ith level NeighborMap\_i from H;
  + check if that level is empty --> terminate when null entry found
  + For (j=0; j<baseofID; j++) {
    - //Fill in jth level of neighbor map
    - While (Dist(N, NM\_i(j, neigh)) > min(eachDist(N, NM\_i(j, sec.neigh)))) {
      * neigh=sec.neighbor;
      * sec.neighbors=neigh−>sec.neighbors(i,j);
    - }
  + }
  + H = NextHop(i+1, new\_id);
* }
  + - Route to current surrogate via new\_id;
    - Move relevant pointers off current surrogate;
    - Call notifyneighbors(surrogate(new\_id))

}

contactGatewayNode(){

* Get a node from supervisor that is not yourself
  + surrogate root
  + Returns Node G
* Returns Node G’s id

}

routeToObject(OG) {

* Return root node of where object is (or would be) located
* Uses nextHop function?
* “A route to a non-existent identifier will encounter empty neighbor entries at various positions along the way. In these cases, the goal is to select an existing link which acts as an alternative to the desired link (i.e. the one associated with a digit of I). This selection is done with a deterministic selection among existing neighbor pointers. Routing terminates when a neighbor map is reached where the only non-empty entry belongs to the current node. That node is then designated as the surrogate root for the object.”

}

updateYourNeighborMap{

* While (Dist(N, NM\_i(j, neigh)) > min(eachDist(N, NM\_i(j, sec.neigh)))) {
  + - neigh=sec.neighbor;
    - sec.neighbors=neigh−>sec.neighbors(i,j);
* }

“After repeating this process for each entry, we have a near optimal neighbor map. The neighbor map population phase requires each neighbor map to be optimized in this manner until there are no nodes to put in the map, due to network sparsity.

The new node stops copying neighbor maps when a neighbor map lookup shows an empty entry in the next hop. It then routes to the current surrogate for *new\_id*, and moves data meant for *new \_id* to *N*.

}

sendHello(new\_neighbor, N){

Node N sends hello to Neighbor new\_neighbor H(i)

}

receiveHello(new\_neighbor, N){

* SendNeighborMap()
* “When we proceed to fill in an empty entry at *N*, we know from our algorithm the range of objects whose surrogate route were moved from” [N+1 entry location]
* “We can then explicitly delete those entries”
* “republish those objects”
* “establishing new surrogate routes which account for the new inserted node.”

}

sendNeighborMap(new\_neighbor, N){

Neighbor new\_neighbor sends its neighbor map to Node N

}

optimizeNeighborMap(){

“Optimizing means comparing distances between N and each neighbor entry and its secondary neighbors. For any given entry, if a secondary neighbor is closer than the primary neighbor, then it becomes the primary neighbor”

* N calls LookupNeighborMap()

}

LookupNeighborMap() {

“Looks up nodes in its neighbors’ neighbor maps, and compares its distance to each of them to determine if they are better potential neighbors. This optimization repeats until no significant improvement can be made by looking for further neighbors.”

}

notifyNeighbors(surrogate(new\_id)){

Purpose: notifies nodes who have an empty entries where *N* should be filled in

* Call notifyEmpties();
* for all neighbors and secondary neighbors in each level
  + Call Hello() ;

**}**

notifyEmpties(){

Purpose: “Use surrogate(new\_id) backptrs to notify nodes by flooding back levels to where surrogate routing first became necessary.

* traverse the surrogate’s backpointers back level by level to the level where surrogate routing first became necessary.
  + Call Hello()

}

publishObject (OG) {

* Publish, or make available, object on the local node. This call is best effort, and receives no confirmation.
* A server , storing an object O (with GUID OG , and root OR3), periodically advertises or publishes this object by routing a publish message toward OR.
* In general, the nodeID of OR is different from OG, OR is the unique [2] node reached through surrogate routing by successive calls to NEXTHOP(\*, OG).
* Each node along the publication path stores a pointer mapping, < OG , S> , instead of a copy of the object itself. When there are replicas of an object on separate servers, each server publishes its copy.
* Tapestry nodes store location mappings for object replicas in sorted order of network latency from themselves.
* A client O locates by routing a message to OR. Each node on the path checks whether it has a location mapping for O.
  + If so, it redirects the message to S.
  + Otherwise, it forwards the message onwards to OR (guaranteed to have a location mapping).

}

unPublishObject (OG) {

Best-effort attempt to remove location mappings for O.

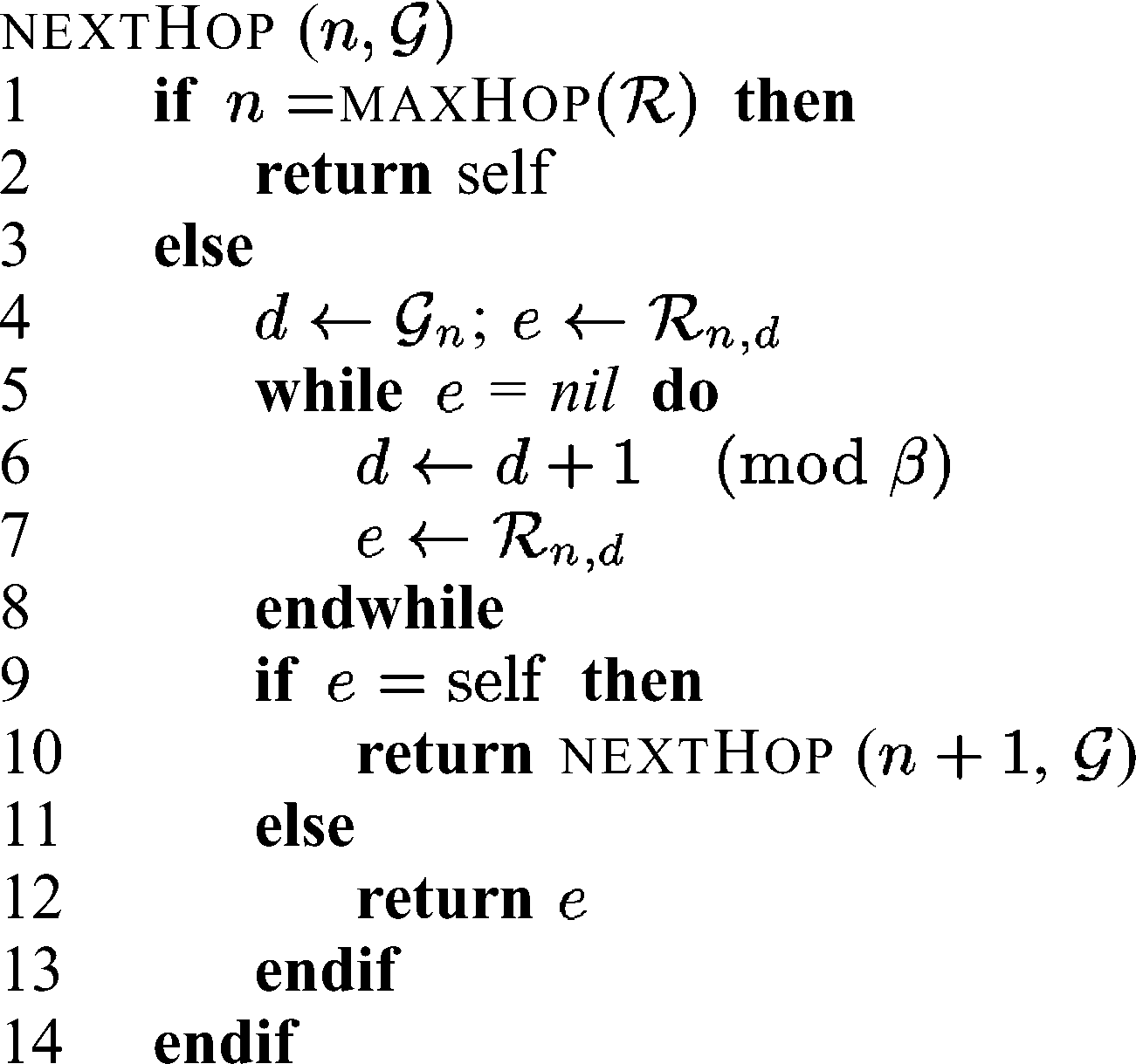
}

routeToNode (N, Exact):{

Route message to application Aid on node N. “Exact” specifies whether destination ID needs to be matched exactly to deliver payload

}

nextHop(n,G){

if n = MaxHop(R) then

return self

else

d <- Gn

e <- Rn,d

while d <- d\_ 1 (modB)

e <-Rn,d

endwhile

if e - self then

return NextHop(n+1, G)

else

return e

endif

endif

}

sendFirst(childPid, *numRequestToSend* ){

send request to

…

new*numRequestToSend = numRequestToSend – 1*

PUBLISHOBJECT(OG, Aid)

Wait one second

sendFirst(childPid, *newnumRequestToSend* )

}