Section 1: Overview of the Algorithm

Our design of a new distributed mutual exclusion algorithm is both quorum-based (like Maekawa’s algorithm) and token-based (like Raymond’s algorithm). Our algorithm operates as follows, on a high-level:

* Initialization phase:
  + Discovers the number of nodes in the system from its configuration file (pre-defined)
  + Performs quorum assignments following the non-null intersection rule for each quorum pair defined in Maekawa’s paper
  + Assigns a privilege node
  + Privilege node releases privilege
* Execution phase:
  + Algorithm accepts REQUEST messages; enforces mutual exclusion
* Termination phase:
  + Algorithm terminates
  + Resource clean-up

Paper notation: pseudocode is annotated in *italics* text.

Section 2: Algorithm Design

This section details each step of the algorithm. We assume the following for our algorithm:

* Network is lossless – messages will eventually be delivered although not necessarily in the order they were sent
* Nodes are completely reliable

Part 1: Initialization Phase: Number of Nodes

The number of nodes in the system must be predefined. The algorithm discover the number of nodes in the system during its initialization phase, by reading the system configuration file. Details follow in the Testbed design section.

Part 2: Initialization Phase: Node Variables

Each Node is responsible for keeping track of the following variables, defined with respect to Node X:

* QUORUM (fixed-sized array): list of every member in Node X’s quorum
* HOLDER (string): indicates the relative position of the node holding the Privilege Token with respect to Node X. Value: “empty”, “self” or a node of Node X’s quorum
* USING (Boolean): indicates if Node X is currently executing its critical section. Values: “true” or “false”.
* REQUEST\_Q (vector): FIFO queue holding the names of the quorum nodes that have sent a REQUEST message to X, including Node X itself.

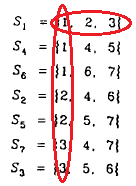
During the initialization process, each Node will set the following for each variable:

* QUORUM = {x,y,z,..}
* HOLDER = empty
* USING = false
* REQUEST\_Q = empty

Part 3: Initialization Phase: Quorum Assignment

During the initialization phase of the algorithm, and after the number of nodes in the system is discovered, the algorithm assigns a quorum to each node. This quorum assignment occurs in the manner defined by Maekawa in his paper. As a restriction, the maximum quorum size is 2\*sqrtN.

Below is an example of the quorum assignment for a seven node system.



TBD – need pseudocode for quorum assignment algorithm

Part 4: Initialization Phase: Node 1 Receives PRIVILEGE Message

When the algorithm is in its initialization phase, it will assign one node in the system to be the privilege node, the node holding the token which gives the right to execute a critical section.

For simplicity, the algorithm will always assign Node 1 as the privilege node during its initialization phase.

Node 1 simulates receiving a PRIVILEGE message. It sets its HOLDER variable to “self”. It sends a HOLDER\_UPDATE(1) message to every member in its quorum to notify them that it is the current token holder. It checks its REQUEST\_Q (which is empty) and does nothing.

Part 6: Mutual Exclusion Algorithm

Sending a REQUEST Message

When Node X invokes mutual exclusion, it first adds its own message to its REQUEST\_Q. It then sends the REQUEST message to the HOLDER of the privilege token; if it does not know who the HOLDER is, it will send a REQUEST to every member of Quorum X.

//Node X is sending the REQUEST(x) message

add REQUEST(X) msg to X.REQUEST\_Q;

if X.HOLDER != “empty” {

send REQUEST(X) to X.HOLDER;

}

else {

for each member in X.QUORUM {

//Other than the node itself

send REQUEST(X);

}

}

Receiving a REQUEST Message

When a member of a QUORUM (say, Node Y) receives a REQUEST(X) message, it checks to see if it knows who the HOLDER node is. If it is the HOLDER node, it sends a PRIVILEGE message back to Node X. If it is not the HOLDER node, but knows who is, it adds REQUEST(X) to its REQUEST\_Q and sends the REQUEST message to the HOLDER node. If it does not know who the HOLDER node is, it ignores the REQUEST message.

//Node X has sent the REQUEST message

//Node Y is receiving the REQUEST message

if Y.HOLDER != “empty” {

if Y.HOLDER == “self” && Y.USING == “true” {

add message to Y.REQUEST\_Q;

}

else if Y.HOLDER == “self” && Y.USING == “false” {

set Y.HOLDER = X;

send PRIVILEGE message to X;

}

else {

add REQUEST(X) message to Y.REQUEST\_Q;

send REQUEST(Y) to HOLDER;

}

}

else {

//message is ignored since the receiving node does not know who the HOLDER is

}

Receiving a PRIVILEGE Message

When Node X receives a PRIVILEGE message, it immediately sends an UPDATE\_HOLDER(X) message to all of its quorum members (but not to the node than sent X the PRIVILEGE message). It then checks the oldest entry in its REQUEST\_Q. If this entry is “self”, then Node X enters its CRITICAL SECTION. If this entry is another node in its quorum, it deletes this entry from its REQUEST\_Q, and sends the PRIVILEGE message to the other Node. It also updates its HOLDER variable to the Node it sent the PRIVILEGE message to.

//Node Y sent PRIVILEGE message

//Node X is receiving PRIVILEGE message

for each member in X.QUORUM != Node Y {

send UPDATE\_HOLDER(X);

//no need to send UPDATE\_HOLDER message to Node Y

//since it is the node that sent the PRIVILEGE message

}

MSG = pop X.REQUEST\_Q;

if MSG != “empty” {

if MSG == “self” {

set X.USING == “true”;

CRITICAL\_SECTION();

set X.USING == “false”;

MSG = pop REQUEST\_Q;

if MSG != “empty” {

X.HOLDER = MSG node;

send PRIVILEGE message to MSG node;

}

}

else {

X.HOLDER = MSG node;

send PRIVILEGE message to MSG node;

}

}

Receiving a UPDATE\_HOLDER(X) Message

When a node receives an UPDATE\_HOLDER(X) message from one of its quorum members, it sets its HOLDER variable to X.

NODE.HOLDER = X;

Exiting Critical Section

When a node exits its CRITICAL\_SECTION, it will set its USING variable to false. Then it must check its REQUEST\_Q to see which node is requesting the privilege, if any. If there is more than one message in REQUEST\_Q, it will forward all of these messages to the next privilege node as well.

//Node X is exiting its CRITICAL\_SECTION

X.USING = “false”;

MSG = pop X.REQUEST\_Q;

if MSG != “empty” {

set X.HOLDER = MSG node;

send PRIVILEGE to MSG node;

for each message remaining in X.REQUEST\_Q, if any{

send REQUEST(X) message to MSG node;

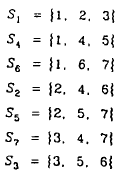
}

}

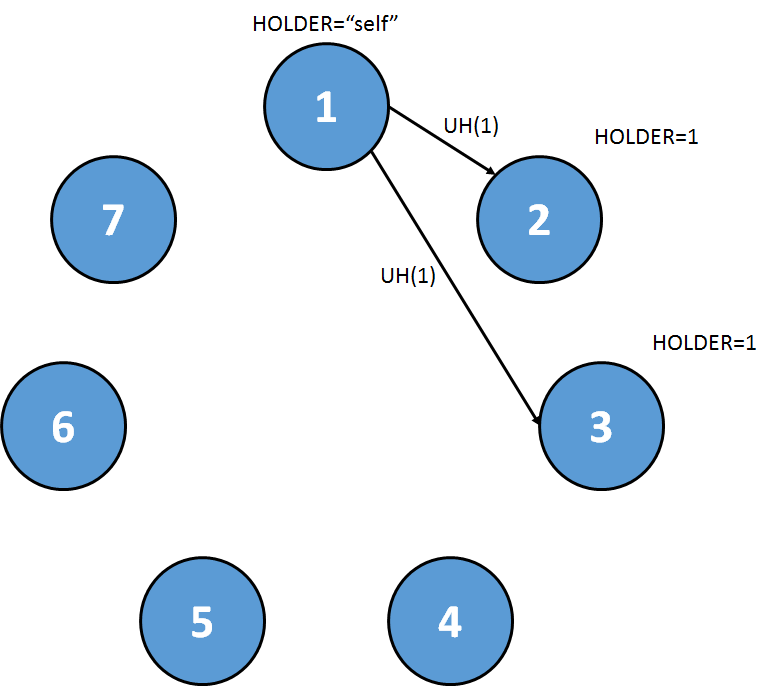
//else, do nothing, wait for a REQUEST message

Algorithm Example

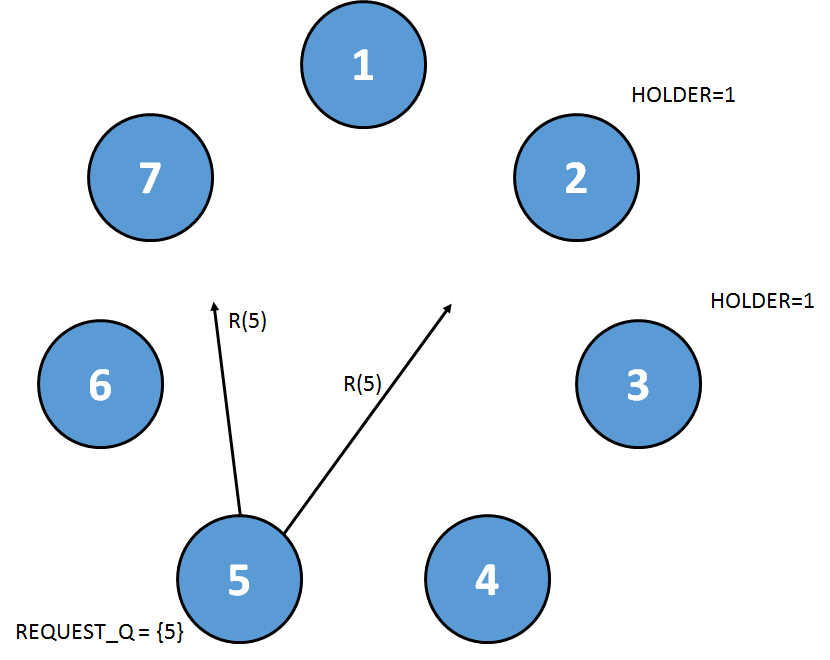
For this example, there are seven Nodes in the system. Thus, there are seven quorums, each with a quorum size of three (see below).



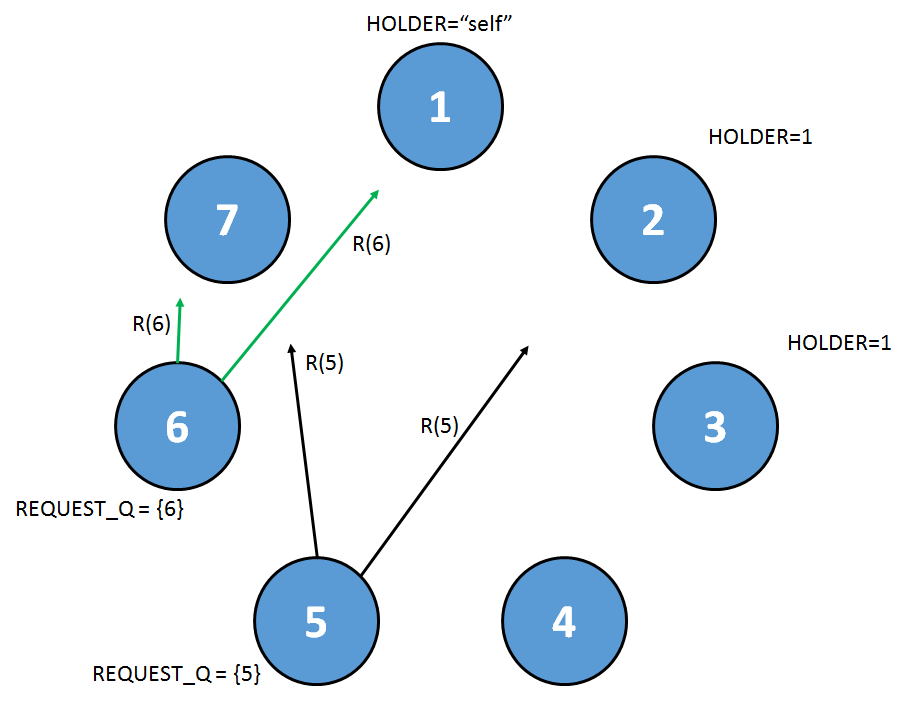
Initialization Step: Node 1 is HOLDER. It sets 1.HOLDER=”Self”. It sends out an UPDATE\_HOLDER message to each member of its quorum (Node 2 and 3). Upon receipt of UPDATE\_HOLDER message, Nodes 2 and 3 set 2.HOLDER=1 and 3.HOLDER=1 respectively.



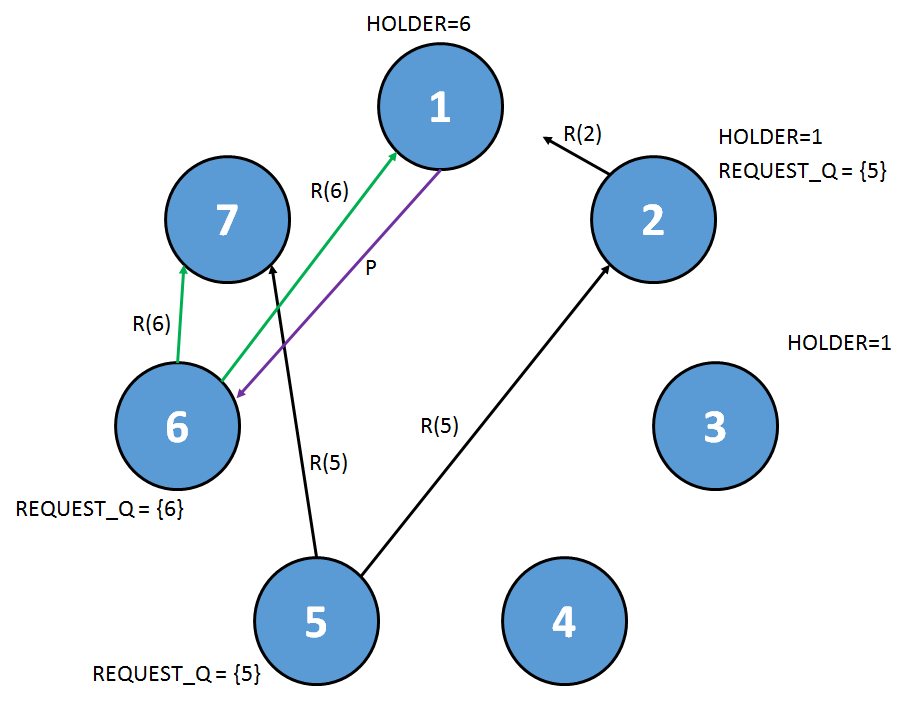
Step 1: Node 5 wants to send a REQUEST message. It does not know who the HOLDER is. It adds 5 to its REQUEST\_Q, and sends out REQUEST(5) to each of its quorum members (Node 2 and 7).



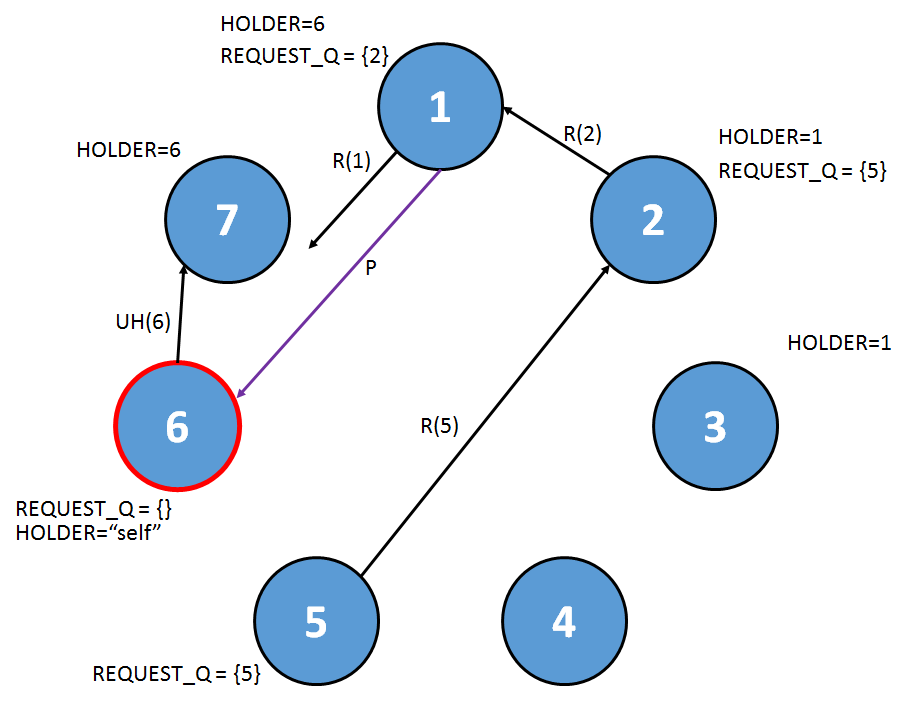
Step 2: At the same time, Node 6 wants send a REQUEST message. It does not know who the HOLDER is. It adds 6 to its REQUEST\_Q, and sends out REQUEST(6) to each of its quorum members (Node 1 and 7).



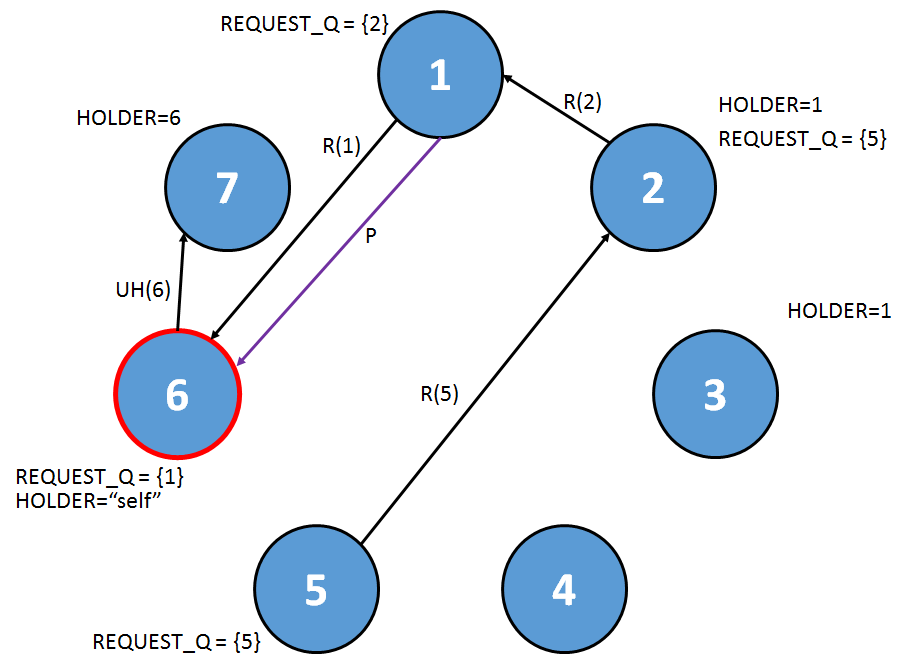
Step 3: Node 7 receives both R(5) and R(6). It does not know who the holder is so the messages are ignored. Node 2 receives R(5). It knows who the holder is, so it adds R(5) to its REQUEST\_Q, and sends a REQUEST message to Node 1. At the same time, Node 1 receives R(6). It is the HOLDER, so sets its HOLDER=6, and sends a PRIVILEGE message to Node 6.



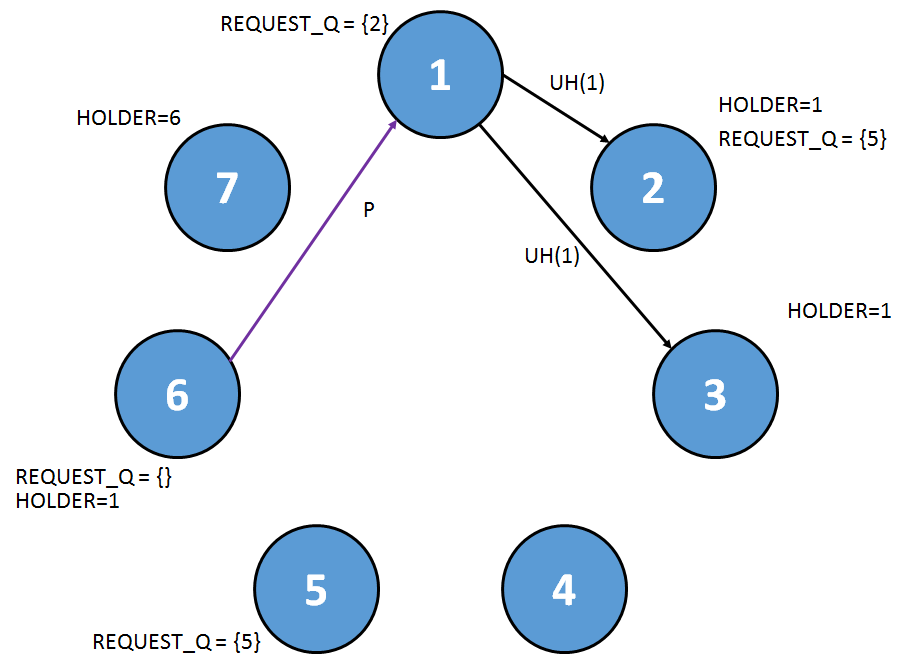
Step 4: Node 6 receives a PRIVILEGE message. It sends out an UPDATE\_HOLDER(6) message to all its quorum members (except the member that just sent the PRIVILEGE message, Node 1 in this case) – so UH(6) is sent to Node 7. Its REQUEST\_Q indicates that the privilege is for itself. It deletes its entry in the REQUEST\_Q, and enters CRITICAL\_SECTION. At the same time, Node 1 receives the R(2) message, sent on behalf of Node 5. Node 1 knows that the HOLDER is Node 6, so it adds R(2) to its REQUEST\_Q and sends a REQUEST(1) message to Node 6.



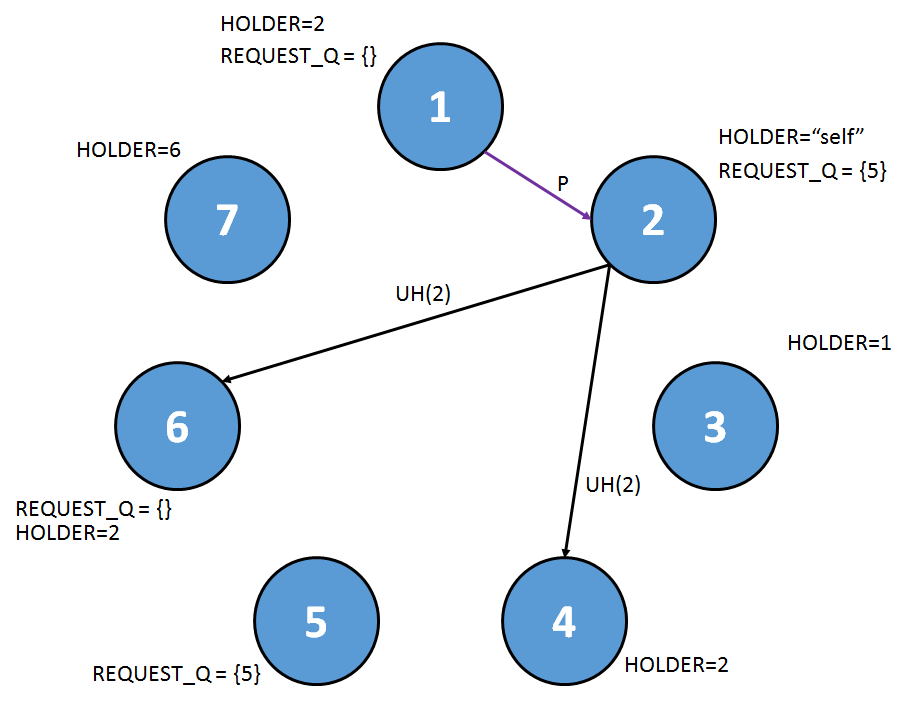
Step 5: Node 6 receives R(1) while in its CRITICAL\_SECTION. It adds R(1) to its REQUEST\_Q.



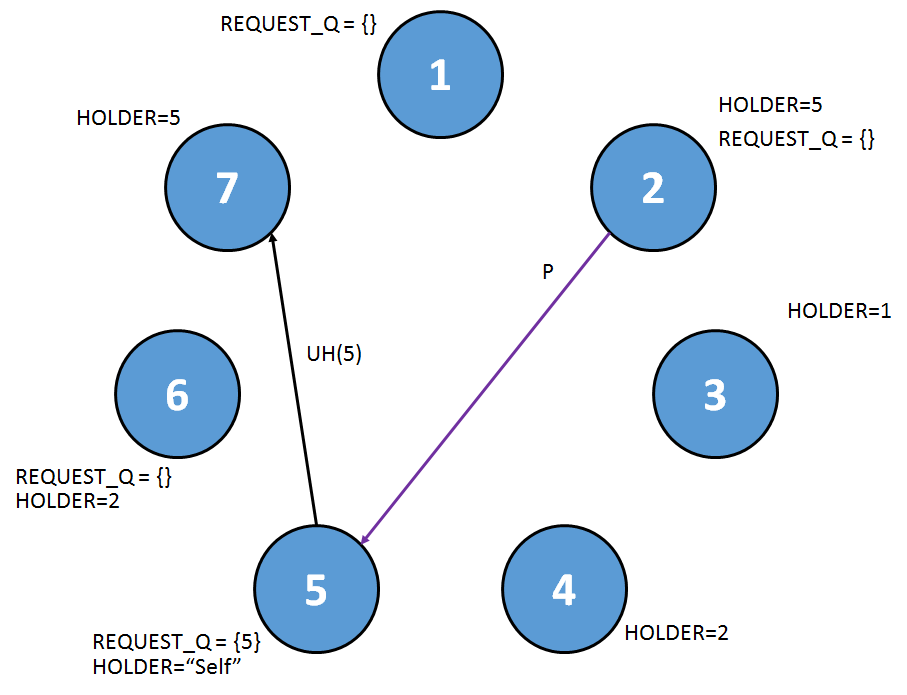
Step 6: Node 6 exits its CRITICAL\_SECTION. It checks its REQUEST\_Q and sees the REQUEST message from Node 1. It deletes the message from its REQUEST\_Q, sets HOLDER=1, and sends the PRIVILEGE message to Node 1. Node 1 receives the PRIVILEGE message and sets HOLDER to “self”. It also sends an UPDATE\_HOLDER to Node 2 and Node 3 (its quorum members).



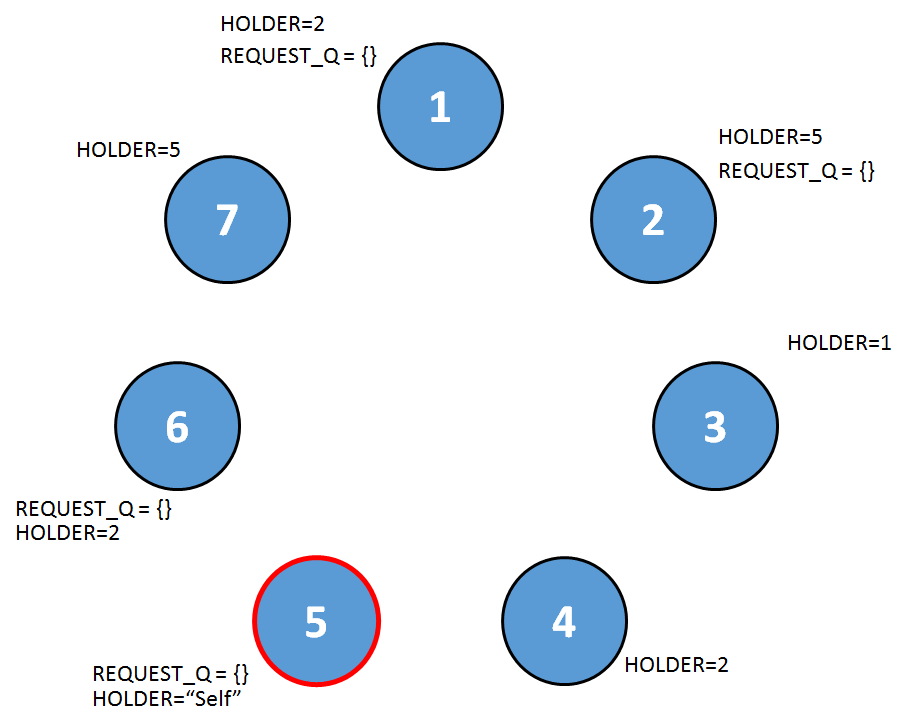
Step 7: Node 1 processes the PRIVILEGE message. It sees that Node 2 requested privilege. It deletes Node 2 from its REQUEST\_Q, sets HOLDER=2, and sends the PRIVILEGE message to Node 2. Node 2 receives the PRIVILEGE message, and sends UPDATE\_HOLDER(2) messages to all of its quorum members (Node 4 and Node 6).



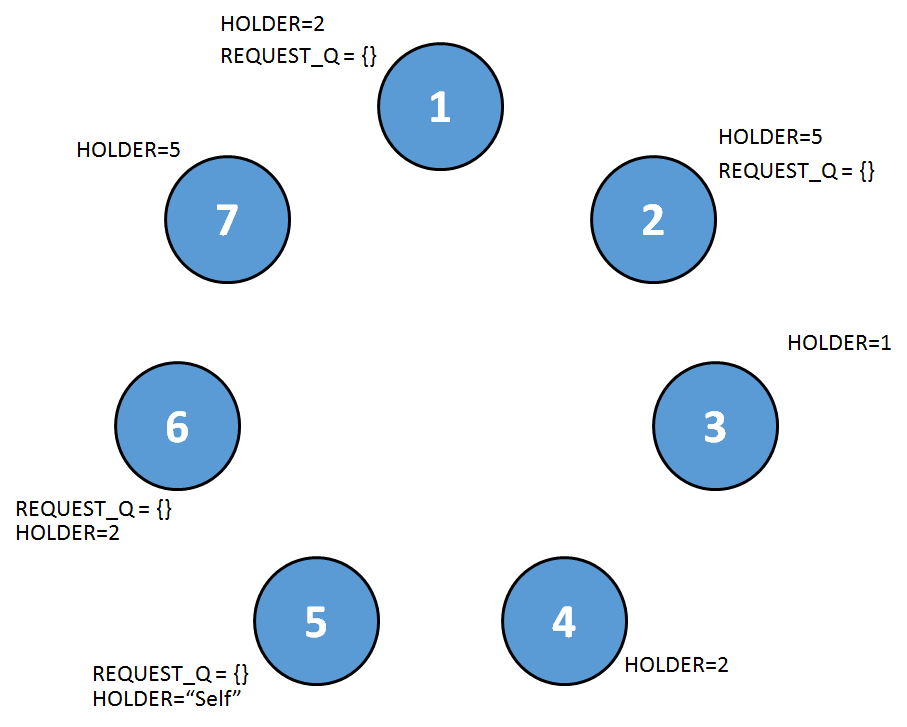
Step 8: Node 2 checks its REQUEST\_Q, and notices that the privilege message is for Node 5. It removes Node 5 from its REQUEST\_Q, it sets HOLDER=5, and it sends the PRIVILEGE message to Node 5. Node 5 receives the PRIVILEGE message and sends an UPDATE\_HOLDER(5) message to its quorum members (only Node 7 since Node 2 sent the message).



Step 9: Node 5 processes the PRIVILEGE message. It checks its REQUEST\_Q and notices the PRIVILEGE message is for itself. It removes Node 5 from its REQUEST\_Q, and enters its CRITICAL\_SECTION.



Step 10: Node 5 exists its CRITICAL\_SECTION. It has an empty REQUEST\_Q, so it just awaits a REQUEST message.



Part 7: Algorithm Termination

TBD

Section 3: Testbed Design

Our testbed is built to support running Maekawa’s algorithm as well as our own. The following sections detail testbed design specifications.

Part 1: Configuration File

Read config file to help algorithm through its initialization phase

TBD

Part 2: Algorithm Selection

Running our algorithm or Maekawa’s algorithm

TBD

Part 3: Testbed Design and Diagram

TBD

References

1. Maekawa, Mamoru. (May 1985) “A sqrtN Algorithm for Mutual Exclusion in Decentralized Systems”. *ACM Transactions on Computer Systems*, Vol. 3 (No 2), pp. 145-159.
2. Raymond, Kerry. (February 1989) “A Tree-Based Algorithm for Distributed Mutual Exclusion”. *ACM Transactions on Computer Systems*, Vol. 7 (No 1), pp. 61-77.