Lab # 2

Peer-to-Peer Storage in the RedNet Distributed System

COMP 420, COMP 532

1 Introduction

In this project, you will design and implement a simple peer-to-peer storage system for programs running on the RedNet distributed system. The particular peer-to-peer system used in this project is roughly similar to the Pastry system as discussed in class, but it is simpler than standard Pastry and also more appropriate in the smaller network size that can be simulated for projects in this class. The RedNet peer-to-peer storage system consists of two components: the *routing overlay protocol* and the *storage system* that makes use of this overlay.

2 The RedNet Peer-to-Peer Routing Overlay Protocol

The routing protocol used in the RedNet peer-to-peer overlay is essentially equivalent to what the textbook describes as *Stage 1* in its description of the Pastry protocol. In particular, in the RedNet peer-to-peer routing system, only *leaf sets* are used, without Pastry's separate routing table. This section gives an overview of the operation of the RedNet peer-to-peer routing overlay protocol.

2.1 Node Routing State

Each computer (i.e., node) participating in the peer-to-peer routing overlay protocol is assigned a 16-bit *nodeID* that identifies the node within the overlay. The nodeIDs are considered to be *unsigned*, and the nodeID space is considered to be *circular*, such that the nodeID immediately following nodeID $2^{16} - 1$ is nodeID 0. A nodeID variable should be declared as type `nodeID`, as defined in the include file `rednet-p2p.h`, and a user process may get its own nodeID by calling the procedure `nodeID GetNodeID(void)`.

Each participating peer-to-peer node maintains a *leaf set* listing information about other nodes whose nodeIDs are numerically closest to this node's own nodeID. The size of a node's leaf set should be `P2P_LEAF_SIZE` entries, with half of these entries representing nodes whose nodeIDs are *less than* this node's nodeID, and half representing nodes whose nodeIDs are *greater than* this node's nodeID. The value of `P2P_LEAF_SIZE` is given in the include file `rednet-p2p.h`.

Each entry in a node's leaf set should contain the following information for each corresponding node:

- The peer-to-peer *nodeID* for the node represented by this entry.
- The *RedNet process ID* of that process. Using this process ID, a RedNet message can be sent directly to that process using the RedNet `SendMessage` operation.

2.2 Joining the Peer-to-Peer Overlay

For a new node to join the peer-to-peer overlay, it must first identify some existing node that is already part of the overlay. To do so, the new node should use an *expanding ring search* to explore other RedNet nodes near the new node (where "near" is relative to the network topology links between directly connected computers in the RedNet distributed system). As defined in the separate specification "*The RedNet Distributed System Programming Environment*," a node can broadcast a message to all of the directly connected

RedNet computers using `TransmitMessage` with a destination process ID of -1. More details on how to implement an expanding ring search are provided below in Section 4.3.

After identifying an existing node that is part of the peer-to-peer routing overlay, the new node sends a special *join* message to this existing node, containing (within the body of the message) the RedNet process ID of the new node that is joining the peer-to-peer overlay. The peer-to-peer destination nodeID of this *join* message should be the new node's own nodeID.

This existing node then routes the *join* message using the peer-to-peer overlay, such that, as described below in Section 2.3, the *join* message is delivered to the existing peer-to-peer node whose nodeID is numerically closest to the new node's nodeID (the new node is *not* yet a part of the peer-to-peer overlay, so the *join* message will not be delivered to the new node itself).

That existing node to which the *join* message is delivered should then return a message directly to the new node, using the new node's RedNet process ID, containing the existing node's nodeID and its complete leaf set. The new node then uses this information as the basis for creating its own leaf set.

2.3 Routing using the Peer-to-Peer Overlay

When routing a message using the peer-to-peer overlay, the message is ultimately delivered to the node whose nodeID is equal to the destination nodeID, or if no existing node is assigned that nodeID, the message is delivered to the existing node whose nodeID is numerically closest to the destination nodeID. As described above in Section 2.1, the nodeIDs are considered to be *unsigned*, and the nodeID space is considered to be *circular*, such that the nodeID immediately following nodeID $2^{16} - 1$ is nodeID 0. In determining which existing nodeID is numerically closest to the destination nodeID, if two nodeIDs are equally close (one less than the destination nodeID and the other greater than the destination nodeID), the one less than the destination nodeID is chosen.

Through the routing protocol, the message progresses gradually closer to the destination. In particular, in routing a message using the peer-to-peer overlay, a node determines the next node to which to forward the message by examining its own leaf set. The node then uses `SendMessage` to forward the message to the node in the leaf set whose nodeID is numerically closest to destination nodeID, using that node's RedNet process ID from the leaf set entry.

However, if in examining its own leaf set, the node determining the next node to which to forward the message determines that *this node itself* is the node whose nodeID is closest (no node listed in this node's leaf set has a nodeID closer than this node's own nodeID), then the message is not forwarded further. Rather, the message is delivered locally at this current node, as it is the existing node whose nodeID is numerically closest to the destination nodeID.

2.4 Maintaining Peer-to-Peer Overlay State

In order for a node to learn of changes in the nodes within the peer-to-peer network, the node must periodically exchange its own nodeID and the current contents of its leaf set with each other node listed in its leaf set. This exchange allows the node to learn of the addition of new nodes or the departure of existing nodes that may require a change in the node's own leaf set.

3 The RedNet Peer-to-Peer Storage Protocol

Making use of the peer-to-peer routing overlay described above, the RedNet peer-to-peer network also supports a file storage system similar to the PAST system that runs over Pastry, as described in class. Each file is identified by a unique *fileID*, and a fileID variable should be declared as type `fileID`, as defined in the include file `rednet-p2p.h`.

The files stored may be of any size up to `P2P_FILE_MAXSIZE` bytes, as defined in `rednet-p2p.h`. Files, once stored, may not be modified, although a stored file may later be deleted.

The RedNet peer-to-peer storage system must support the following four operations, *and you must implement the following four procedures with this interface*:

- `int Join(nodeID)`

  This operation is used by the user process to request this computer in the RedNet distributed system to join the peer-to-peer storage system. The peer-to-peer nodeID to use for this node is given by `nodeID` argument; the user process may get this number to pass as this argument by calling `GetNodeID`, as described in Section 2.1.

- `int Insert(fileID fid, void *contents, int len)`

  This operation is used to store a file in the peer-to-peer storage system. The file's fileID is given by `fid`, the pointer `contents` points to a location in memory holding the file's contents, and `len` gives the length in bytes of that contents.

  To implement the *insert* operation, given the file's fileID and file contents, a message containing a copy of the file should be routed using the peer-to-peer routing overlay from the node requesting the *insert* operation to a destination nodeID equal to the file's fileID; the request message will be delivered to the node A whose nodeID is numerically closest to this nodeID. This A node should store a copy of the file.

  In addition, this node A should also send a copy of the file directly to the 2 other nodes, B and C, in A's leaf set (i.e., thus not including the node A itself, since a node is not listed in its own leaf set) whose nodeIDs are numerically closest to the file's fileID. Each of these 2 nodes, B and C, should also store a copy of the file and should then reply directly to node A confirming the result.

  Once completed, the node A to which the original request message was delivered should finally send a reply message directly to the node that initiated the *insert* operation, confirming the result.

- `int Lookup(fileID fid, void *contents, int len)`

  This operation is used to retrieve a copy of a file that has previously been stored in the peer-to-peer storage system. The file's fileID is given by `fid`, the pointer `contents` points to a location in memory where the file's contents should be placed upon a successful lookup, and `len` gives the length in bytes of that buffer. If the file's contents are longer than `len` bytes, only the first `len` bytes of the file should be retrieved.

  To implement the *lookup* operation, given the file's fileID, a message containing the request should be routed using the peer-to-peer routing overlay from the node requesting the *lookup* operation to a destination nodeID equal to the file's fileID; the message will be delivered to the node A whose nodeID is numerically closest to this nodeID. Node A should then reply directly to the node that initiated the *lookup* operation, confirming the result and returning a copy of the file if successful.

- `int Reclaim(fileID fid)`

  This operation is used to attempt to remove a file that has previously been stored in the peer-to-peer storage system and to reclaim the storage space currently used by the stored copies of that file. The file's fileID is given by `fid`. For simplicity, the *reclaim* operation does not guarantee that all copies of the file have been deleted but attempts to achieve this result without the expense and complexity a true guarantee would require.

To implement the *reclaim* operation, given the file's fileID, a message containing the request should be routed using the peer-to-peer routing overlay from the node requesting the *reclaim* operation to a destination nodeID equal to the file's fileID; the request message will be delivered to the node A whose nodeID is numerically closest to this nodeID. This node should delete its copy of the file.

In addition, this node A should also send a request directly to the 2 other nodes, B and C, in A's leaf set (i.e., thus not including the node A itself, since a node is not listed in its own leaf set) whose nodeIDs are numerically closest to the file's fileID. Each of these 2 nodes, B and C, should also delete their stored copy of the file and should then reply directly to node A confirming the result.

Once completed, the node A to which the original request message was delivered should finally send a reply message directly to the node that initiated the *reclaim* operation, confirming the result.

4 Implementation Details

4.1 Division of Responsibility between User Process and System Kernel

For this project, the peer-to-peer routing overlay as described in Section 2 should be implemented entirely within the operating system kernel, with the peer-to-peer storage system described in Section 3 that utilizes this routing overlay also implemented within the kernel, with the exception of the requests described in Section 3 being initiated by user processes.

In particular, a user process utilizing the peer-to-peer storage system makes requests to the system by sending a message using `SendMessage` to the local operating system kernel; this can be done by sending the message to a destination process ID of 0. The result of such a request is returned by the kernel to the user process by the kernel using `DeliverMessage` to deliver it to the user process. The processing of that request, including all protocol and storage aspects of the request for this project should be implemented within the kernel.

Messages within the peer-to-peer routing overlay are implemented as messages from one operating system kernel to another. Such a message can be transmitted using `TransmitMessage`.

Within the operating system kernel, all aspects of your implementation of the peer-to-peer system must be a part of the `HandleMessage` procedure within your kernel, or within separate procedures called by `HandleMessage`. Indeed, it is specifically recommended to *not* include all of your code literally within your `HandleMessage` procedure; it is much better to divide the code into separate procedures called as needed by your kernel's `HandleMessage` procedure.

4.2 Message Formats

You must define your own message formats, both for user process messages and for operating system kernel messages.

For user process messages, you must support at least the following four types of messages, corresponding to the four types of requests as described in Sectionstorage:

- *join*: This message must be sent from a user process to the local operating system kernel to request the node to join the peer-to-peer routing overlay. Until the kernel receives and processes this message, the kernel should otherwise *handle all messages normally* (including those that are part of a flood, as described below in Section 4.3) *other than those messages that require participation in the peer-to-peer routing overlay*.

- *insert*: This message must be sent from the user process to the local operating system kernel to request an *insert* operation, as described above in Section 3.

- *lookup* This message must be sent from the user process to the local operating system kernel to request a *lookup* operation, as described above in Section 3.

- *reclaim* This message must be sent from the user process to the local operating system kernel to request a *reclaim* operation, as described above in Section 3.

In defining the format for such user process messages, it is recommended you define a `struct` to represent a common header format for all of these messages. For example, include a field at the beginning of the `struct` to contain a number indicating the type operation represented by the message (e.g., 0 for join, 1 for insert, etc.), followed by a field of type `fileID` to carry the fileID and an field of type `int` to carry the length (some of these fields would not be used for some types of requests). By including an additional field of type `int`, you can also use the same `struct` to represent the reply message returned indicating the result of the request. Finally, following such fixed parts of the message format, the message could contain the bytes of the file contents itself, both for an *insert* request and for a *lookup* reply.

Likewise, you must define the format you will use for messages sent between the operating system kernel on different nodes to implement the peer-to-peer routing overlay. As for user process messages, it is recommended you define a `struct` to represent a common header format for such messages, including a field of type `int` to indicate the message type as well as any other fields you need for these kernel messages (these will not be the same as the fields suggested above for user process messages to the kernel). Finally, following the fixed kernel header format you define, if needed you would include the user process message that should be the "payload" of the kernel message.

4.3 Performing a Hop-Limited Flood of a Message

In order to implement an expanding ring search, as discussed above in Section 2.2, you must implement a *hop-limited flood* of a message. The hop-limited flood delivers a message to all nodes within a given number of network "hops" of the originating node, where a network hop is a transmission over a directly connected link in the network topology. A hop-limited flood with a hop limit of 1 hop is directly provided by the broadcast facility of `TransmitMessage` operation with a destination process ID of -1. However, for larger values of the hop limit, you must implement appropriate forwarding of such broadcast messages.

An expanding ring search is performed by first performing a hop-limited flood with a hop limit of 1. If no reply is received from the hop-limit 1 flood, a new flood is initiated from the same node but with a hop limit of 2. If no reply is received from the hop-limit 2 flood, a new flood is initiated from the same node but with a hop limit of 3, and so on until a reply is received.

A hop-limited flood, as described here, is performed between the operating system kernel on the various computers in the RedNet distributed system. In order for a message to be sent by a hop-limited flood, the operating system kernel header of the message must include two fields that are used to control the flood (in addition to any other fields needed for the meaning/use of the message itself):

- A *sequence number*, initialized by the node that originated the flood. The same sequence number (unchanged) must appear in each forwarded copy of the message that is a part of the flood.

- A *hop count limit*, also initialized by the node that originated the flood. The hop count limit in each copy of the message forwarded as a part of the flood is first decremented by the forwarding node, and the message *must not* be forwarded if the new decremented value is 0.

In addition, each node must maintain the following state in order to participate in such a flood:

- A *sequence number*. For each new flood that this node originates, this sequence number is incremented, and the value of the sequence number is used to initialize the sequence number field within the message to be flooded.

- The *maximum sequence number value seen from each other node*. Specifically, for each respective *originating* node from which this node has received a message as part of a flood, this node remembers the *maximum* such sequence number value in any of these messages received from that originating node. The node remembers this maximum sequence number independently for each originating node from which it has received a messages as part of a flood.

Given these fields in each message that is part of a flood and this state maintained by each node, a hop-limited flood of some message proceeds as follows:

- The originating node initializes the sequence number and intended hop limit in the message's header. This node then broadcasts the message to its directly connected neighbors by using `TransmitMessage` with a destination process ID `dest` of -1. The source process ID `src` should be the originating node's own RedNet process ID.

- When any node receives a message as part of such a flood, the node first examines the sequence number in the message. The receipt of the message from the network causes the `HandleMessage` procedure in the operating system kernel on that node to be invoked, and the `src` argument passed to `HandleMessage` identifies the node that originated that flood. If the message sequence number is *less than or equal to* the maximum sequence number previously received in a flood message originated by that `src` node, then the node simply returns from `HandleMessage` without further forwarding the message.

- Otherwise (the message sequence number is *greater than* the maximum sequence number previously received in a flood message originated by that `src` node), then this node remembers that message sequence number as the new maximum sequence number received in a flood message originated by that `src` node.

- The node then processes the message in whatever way is required for the type of message that was received as part of the flood.

- Finally, the node decrements the hop count limit in the message. If the new value of the hop count limit is 0, then the node returns from `HandleMessage` without further forwarding the message. Otherwise (the new value of the hop count limit is still greater than 0), then the node forwards the message by broadcasting it to the node's directly connected neighbors by using `TransmitMessage` with a destination process ID of -1. The source process ID `src` in this call to `TransmitMessage` should be *the same* as the `src` value passed to this call to `HandleMessage` (i.e., *not* this node's own process ID); the `src` process ID thus remains equal to the process ID of the process that *originated* this flood.

Note that a node must participate in forwarding such a flood, even before that node itself has joined the peer-to-peer routing overlay. A flood like this does not use the peer-to-peer overlay.

5 Use of Piazza

If you have not yet registered for this class on Piazza, you should do so. Piazza will allow you to post questions about the course material and to quickly receive answers from me, from the TA, and from your fellow classmates. With Piazza, you can also easily look to see if someone else has already posted a question about a problem that you may also be experiencing. Instead of sending questions by email to me and/or the TA, we encourage you to post your questions on Piazza. You should also check Piazza regularly for new questions, answers, and announcements.

Again, when posting questions or answers about the project on Piazza, please be careful about what you post, to avoid inadvertently violating the course's Honor Code policy described in the course syllabus and below in Section 6. *Specifically, please do not post details about your own project solution, such as portions of your source code or details of how your code works, in public questions or answers on Piazza.* If you need to ask a question that includes such details, please make your question private on Piazza by selecting "Instructor(s)" (rather than "Entire Class") for "Post to" at the top, so that only the course instructor and TA can see your posting.

6 Honor Code Policy

The Honor Code is a special privilege and responsibility at Rice University. As stated in a January 20, 2016 editorial in the *Rice Thresher*: "As incoming students enter Rice, many are surprised by the degree to which the university's Honor Code extends trust to the student body. ... The privileges of the Honor Code stem from the idea that Rice's aim is not just to instill knowledge in its students, but [to] also help them develop moral character. This idea is fundamental to Rice's identity: Students can and should be held to a high moral character standard, and the honor system makes life easier for both students and faculty," for example in making possible freedoms such as take-home exams. I firmly believe in these basic ideas behind Rice's Honor System and trust that you do so as well.

All assignments in this course are conducted under the Rice Honor System, and you are expected to behave in all aspects of your work in this course according to the Rice Honor Code. When in doubt as to whether a specific behavior is acceptable, ask the instructor for a written clarification. Suspected Honor Code violations in this course will be reported in complete detail to the Rice Honor Council. For more information on the Rice Honor System, see `http://honor.rice.edu/` .

Specifically, for each of the programming projects in this course, *the project must be done individually.* During each programming project, students are encouraged to talk to each other, to the TA, to the instructor, or to anyone else about the assignment. This assistance, however, is limited to general discussion of the problem. *Each student must produce their own solution.* Consulting or copying, in any manner, another student's solution (even from a previous class or previous year) is prohibited, and submitted solutions may not be copied, even in part, from any source.

7 Lab 2 Project Submission

Your completed Lab 2 project is due by **11:59 PM on Friday, December 2**, the last day of classes for the semester.

Before submitting your project, please create a file named "`README`" in the same directory as the rest of your files for the project. In this file, please describe anything you think it would be helpful to know in grading your project. This might, for example, mean describing unusual details of your algorithms or data structures, and/or describing the testing you have done and what parts of the project you think work (or don't work).

To submit your Lab 2 project for grading when you are ready, please perform the following two steps:

- First, on CLEAR, change your current directory to the directory where your files for the project for grading are located. For example, use the "`cd`" command to change your current directory. When you run the submission program, it will submit everything in (and below) your current directory, including all files and all subdirectories (and everything in them, too).

- Second, on CLEAR, run the submission program

```
/clear/courses/comp420/pub/bin/lab2submit
```

This program will check with you that you are in the correct directory that you want to submit for grading, and finally, will normally just print "`SUCCESS`" when your submission is complete. If you get any error messages in running `lab2submit`, please let me know.

After your submission, you should also receive an email confirmation of your submission, including a listing of all of the individual files that you submitted.

You may (if you want to) submit your project multiple times or at any time. The `lab2submit` program will only remember the single *most recent* submission you make for this project.