CSc 422, Program 3: Distributed Hash Tables

Due date: December 8th, 2020 at 11:59pm. **No late assignments will be accepted.**

In this assignment we will implement a (simplified) distributed hash table (DHT), which provides a lookup service that is similar in spirit to a traditional hash table. That is, a DHT stores (key,value) pairs, where the key determines which of a set of nodes stores the associated value.

We are providing you with some skeleton code to help get you started. Please download: `http://dkl.cs.arizona.edu/teaching/csc422-fall20/assignments/csc422-prog3Files.tar.gz`.

The specifications and rules for this assignment are as follows.

1. You will use MPI to implement communication between the DHT nodes. Execute your program with `mpirun -np <numProcesses> <executable>`.

   - Note that you will not use any -host or -hostfile option. MPI will run each rank in a separate process on the local machine.
   - Rank 0 will act as the *head* node and will not store any data. It will receive commands from the *command node*, which will be rank $numProcesses - 1$. The command node will not be in the set of storage nodes. The head node will forward commands as necessary to storage nodes. Also, the head node will be responsible for sending the results of a `GET` command back to the command node.
   - Ranks 1 through $numProcesses-2$ will act as storage nodes. Each active storage node maintains a unique storage id between 1 and 1000 (see the `ADD` command below). Rank $numProcesses - 2$ is created upon startup and has a storage id of 1000; it will never be removed. Ranks 1 through $numProcesses - 3$ are inactive until explicitly added to the DHT by an `ADD` command.

2. The command node can send the following commands to the head node via MPI messages. There will be no error checking needed on this assignment—all of our requests will be both syntactically and semantically correct. All commands will be sent as MPI messages; the command will be placed in the MPI message tag (between 0 and 4, inclusive), and the parameters will be sent in the order they appear above (all commands are between zero and two integers).

   - `PUT: 0 <key> <value>`: stores a value on one of the storage nodes. The key must be between 1 and 1000.
   - `GET: 1 <key>`: retrieves the value from the correct storage node and sends the value and the storage id of the node to the head node.
   - `ADD: 2 <rank> <id>`: adds the node of the specified rank to the DHT, giving it the specified id (to be used for determining which keys it will store). All ids will be positive integers.

- `REMOVE: 3 <id>`: removes the node with the specified id from the DHT.
- `END: 4`: terminates the program.

After sending a `GET` message, the command node will invoke the following:
`MPI_Recv(answer, 2, MPI_INT, 0, RETVAL, MPI_COMM_WORLD, MPI_STATUS_IGNORE).`
Variable *answer* is an array of two integers, which should contain the value and the storage id (in order). The sender is rank 0 (the head node), and the message tag is RETVAL (5). When this message is received by the command node, the `GET` operation is complete.

After sending a `PUT`, `ADD`, or `REMOVE` message, the command node will invoke the following:
`MPI_Recv(&ignore, 1, MPI_INT, 0, ACK, MPI_COMM_WORLD, MPI_STATUS_IGNORE).`
Variable *ignore* is an integer, which is ignored. The sender is rank 0 (the head node), and the message tag is ACK (6). When this message is received by the command node, the `PUT`, `ADD`, or `REMOVE` operation is complete.

After sending an `END` message, the command node should execute `MPI_Finalize`.

3. DHT structure:

   - Nodes are arranged in a circular structure, according to storage id, from lowest to highest. The child node of the head node is the active storage node with the smallest storage id. The child node of an active storage node is the active storage node with the next-smallest storage id, and so on, except for rank $numProcesses - 2$, whose child is the head node. Each active storage node stores any data whose key is greater than its parent's id, but less than or equal to its own id. Figure 1 contains an example DHT with 3 active storage nodes, each storing some data. For example, key 15 is stored on storage id 20 because 15 is greater than 10, but less than or equal to 20.

     This state can be achieved from many command sequences. One such sequence is as follows:

     ```
     2 1 20
     2 2 10
     0 10 11111
     0 25 12345
     0 15 673
     0 20 1
     0 35 54321
     0 5 99999
     ```

     Another is as follows:

     ```
     2 1 20
     0 25 12345
     0 15 673
     ```

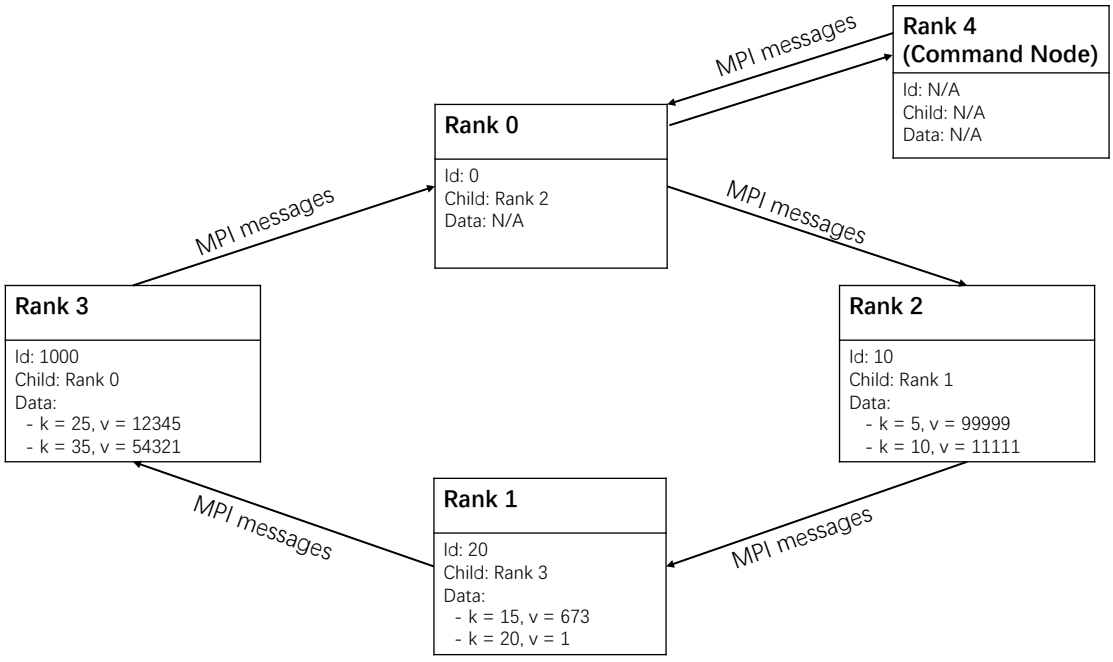

Figure 1: DHT with three active storage nodes, a head node, and the command node; this corresponds to the state of the DHT after the entire command sequence in bullet point 3 on the previous page. MPI is used to communicate between all nodes. The command node sends a command to the head node and then receives the response from the head node once the command has been carried out. Note that communication from the head node to nodes other than its chid is allowed **only** when an `END` message is received.

```
2 2 30
0 35 54321
0 10 11111
3 30
2 2 10
0 20 1
0 5 99999
```

- Storage nodes will exchange data as necessary when nodes are added or removed.
- Nodes may only communicate with their parent and child nodes (the END command is the only exception; see below). When the head node receives a command, it will forward the command to its child (along with any other internal information your program may use to carry out commands). Each storage node will continue to forward the command until it reaches the appropriate node to service it (see specific command descriptions below).
- You *must* ensure that commands are not issued by the command node until the previous

command has *fully* completed. This means that the command node has received the value/storage id pair if the command was `GET`, or the command node has received the acknowledgment (the data is ignored) in the case of `PUT`, `ADD`, or `REMOVE`. For some operations, enforcing this is nontrivial.

4. Command details:

   - `ADD`: The request is forwarded until reaching the correct parent/child of the new node (whether the parent or child does the actual add is up to you). The parent/child then directly contacts the node to be added, which will then initialize itself with the specified id and any other state information you are keeping. If there is any existing data that belongs on the new node, it is moved. Here, that there are no errors means that the command will have a valid rank, and that node will not already be active.
   - `REMOVE`: The request is forwarded until reaching the correct node (which could be the node with the id given, or its parent/child, depending on your implementation). All data on the node to be removed is moved to the appropriate parent/child, parent/child information is updated, the removed node becomes inactive. Here, that there are no errors means that the id specified will always correspond to an existing active node.
   - `PUT`: The request is forwarded until reaching the correct node to store the data (the one with the smallest id that is larger than or equal to the key). Here, that there are no errors means that the same key will never be PUT more than once and that the key will never be larger than 1000.
   - `GET`: The request is forwarded until reaching the node that is storing the data. The node retrieves the value associated with the key and its storage id and circulates both back to the head, who sends it along to the command node. For example, if the command `1 15` was given with the DHT in Figure 1, `673 20` should be sent to the command node.

     The data is not deleted from the node on a `GET` request. Subsequent `GET` operations for the same key should produce the same value, though the the id may change if the data is moved due to an `ADD` or `REMOVE`. Here, that there are no errors means that the key requested will always exist in the DHT.
   - `END`: To guarantee a clean exit from MPI, the head node will send the END command **directly** to every other node (both active and inactive), except for the command node, which just invokes `MPI_Finalize` and then `exit(0)`. Upon receipt of an `END` message, every node must then call `MPI_Finalize` and `exit(0)`.

5. **Important:** Attempts to circumvent the specs will be considered academic dishonesty. Examples of circumventing include: (1) you simply store all keys on the head node, (2) you store all keys on one of the storage nodes, or (3) you maintain all-to-all communication between the DHT storage nodes.

6. Example with comments:

```
input:
2 1 40      // rank 1 is added with id 40
0 15 673    // key 15 is put on storage node 40, (rank 1)
1 15        // key 15 is currently on storage node 40
0 25 12345  // key 25 is put on storage node 40, (rank 1)
2 2 10      // rank 2 is added with id 10
0 35 54321  // key 35 is put on storage node 40, (rank 1)
1 35        // key 35 is currently on storage node 40
0 10 11111  // key 10 is put on storage node 10, (rank 2)
3 40        // storage node 40 (rank 1) is removed; and,
            // keys 15, 25, and 35 are moved to storage node 1000 (rank 3)
1 35        // key 35 is currently on storage node 1000
1 15        // key 15 is currently on storage node 1000
2 1 20      // rank 1 is added with id 20; and,
            // key 15 is moved to storage node 20 (rank 1)
1 15        // key 15 is currently on storage node 20
0 20 1      // key 20 is put on storage node 20 (rank 1)
0 5 99999   // key 5 is put on storage node 10 (rank 2)
1 15        // key 15 is currently on storage node 20
4

output:
673 40
54321 40
54321 1000
673 1000
673 20
673 20
```

7. Here are details of of two MPI operations (in addition to `MPI_Send` and `MPI_Recv`) you need to implement this program.

   - `MPI_Probe`. This operation allows you to "peek" at a message, which means you can determine its contents without actually receiving it.
   - `MPI_Get_count`. This operation allows you to determine the size of a message.

   Both of these operations are in the skeleton code that we have provided. Note that `MPI_Probe` does not receive a message; you still eventually call `MPI_Recv` to do so. Also, the skeleton code uses `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, which allow you to peek the message regardless of sender or tag (and then you can query the value of the sender and the tag). This is useful because you do not know which message you will receive, and from whom (in the general case).

8. You will need to store the key/value pairs on each node. This must be done with a dynamic data structure (e.g., list or hash table) because there is no set maximum number of key/value

pairs per node.

9. Here is our suggested approach to developing this program.

   (a) Learn what you need to for MPI (send and receive, including sending and receiving multiple values in a message) in a *separate* program.

   (b) Understand our skeleton code.

   (c) Implement only the `GET` operation. This means you will have to artificially put key/value pairs on a couple of storage nodes.

   (d) Implement the `PUT` operation.

   (e) Implement the `ADD` operation without redistribution (it will not be fully correct yet).

   (f) Implement the `REMOVE` operation without redistribution (it will not be fully correct yet).

   (g) Implement redistribution for the `ADD` and `REMOVE` operations.

Submission

You must submit `dht.h`, `dht.c`, `dht-helper.h`, `dht-helper.c`, and `command.c`. Note that `command.c`, can contain *only* function `commandNode`. Note that when we test your code, we will be replacing `command.c` with our own version. In addition, function `commandNode` **must** be called within `dht.c` by the command node directly after `MPI_Init`, `MPI_Comm_rank`, and `MPI_Comm_size`, just as it is in the code we are providing you. Do not modify the Makefile (which you need not turn in). You should use the turnin command to submit; the assignment name is cs422-f20-prog3. Please do not submit a tar file—just the files listed.

As with the first two programming assignments, we will be running all tests on lectura. Please make sure your code works there.