**Winter CS454/654 2012**
**Assignment 2**
**Instructor: Bernard Wong**
**Due date: March ~~12~~ 15, 2012  Group size: 2**

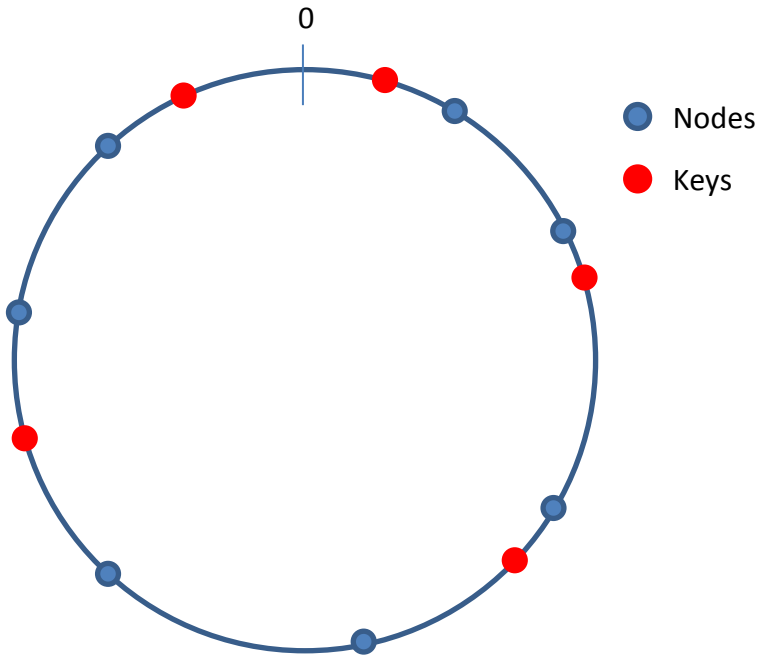
Distributed systems is a rapidly evolving field. Most of the systems we have studied (or will study) in class have stood the test of time; with 20/20 hindsight, we know they each introduced some key new insight to designing scalable, robust, and efficient distributed systems. However, by focusing on seminal systems from the past, the lectures do not sufficiently cover current distributed systems research. Fortunately, this assignment will give us an opportunity to study in detail the design of a modern distributed system. It will also require that your group implement and test this design.

In this assignment, your group will implement a Distributed Hash Table (DHT), a system that is both peer-to-peer (nodes act as both clients and servers) and decentralized (system is self-regulating). A DHT, as one can surmise from its name, provides a very simple and familiar hash table interface: a key/value pair can be stored using a *put* operation, and the value for an associated key can but retrieved using a *get* operation. Also, much like a standard hash table, each key is hashed to determine where it and its associated value should be stored. However, in place of determining hash buckets, the key hashes determine which machines are responsible for storing each key/value pair.

Memcached[1] is a popular distributed object caching system that provides a put/get interface by directly mapping hash buckets to machines. Given a Memcached deployment has *n* machines, each machine is assigned a unique ID from 0 to n -1, and a key/value pair *(k,v)* is stored at machine with ID = *hash(k) mod n.* This design has the tremendous advantage of not requiring any coordination between the Memcached servers. For each key, a client can determine which server to contact by performing *hash(k) mod n* and subsequently sending a get/put query to the corresponding Memcached server. However, there are a number of limitations to this design. A node failure will cause 1/n of the key-space to be inaccessible until the node is repaired or replaced. The deployment is also highly inflexible. Adding a new node will result in changing the key to server mapping function. Nearly all existing key/value pairs will be stored at the wrong server. As Memcached is mainly used as a caching service, this is equivalent to invalidating all previous cache results.

One can avoid this deployment problem by using a technique known as *Consistent Hashing*. Rather than assigning nodes to a fixed number of hash buckets, consistent hashing maps nodes to points on a (long) line or ring. Most systems use rings for the additional symmetry; we will assume nodes are map to points on a ring for the remainder of this assignment. Ideally the points are uniformly distributed along the ring. This can be accomplished by having each node pick a random point, or by using a good hash function, such as SHA-1 or MD5, to generate a value from some unique ID of the node, such as the node's IP address. Collisions are expected to be extremely rare due to the size of the ID space *m*, as we assume that $m >> n$ where *n* is the number of nodes in the system. Similarly, key/value pairs are also mapped to points on the ring using a strong hash function on the key.

[1] http://memcached.org/

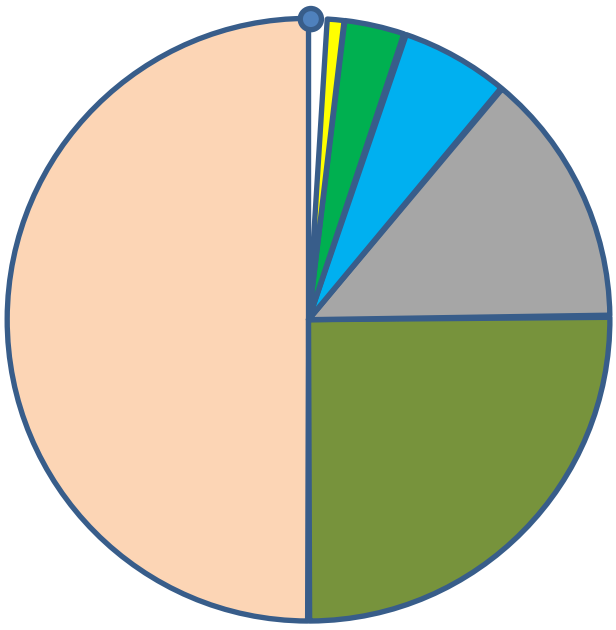



Given that both nodes and key/values pairs are mapped to the same ring, one can deterministically assign key/values to nodes using a distance metric. For example, one method would be to assign a key/value to the numerically closest node (Pastry: A. Rowstron and P. Druschel), or the closest clockwise or counter-clockwise node (Chord: I. Stoica et al.). The distance metric does not even have to be the numeric distance. Kademlia (P. Maymounkov and D. Mazières) uses an XOR metric to determine the distance between points. By using consistent hashing, adding a node *A* would only affect the key/value to node mappings that map to one or both of A's adjacent neighbors. Without key/value replication, removing an existing node from the deployment would not affect any mappings and does not make any of the key-space inaccessible. Those key/values that were stored on the node would no longer be available, but future *puts* to that region of the ring will map to the next closest node rather than to an empty bucket as is the case with Memcached.

With consistent caching, a client (assuming that the nodes in the DHT provide some type of service) can determine which node to issue a get/put request to by hashing the key, and identifying the node that has the closest node ID to the key hash. This requires that the client know the ID of every node in the system, which limits scalability. Alternatively, the client can ask one of the DHT nodes to route the request to the appropriate destination, which requires that each DHT node know the ID of every node in the system. This is better, as we can safely assume that the DHT nodes have a vested interest in providing this service and is willing to pay the overhead required to keep track of the group membership of the system. However, for large systems with hundreds of thousands of nodes, this is still a very challenging task, especially if the rate of node joins/leaves (known as churn) is fairly high.

Instead of requiring each node to deliver get/put requests to the destination in one hop, one can instead allow each request to be routed to the destination over a series of hops. With this more relaxed requirement, a node only needs to know it adjacent neighbors (or just a single neighbor if one fixes the direction to clockwise or counter-clockwise) to route the request along the ring. However, this would require O(n) hops and would likely not provide sufficient performance in a large system. Adding *k* additional neighbors would reduce the expected number of hops, but most naïve schemes would only reduce it by a factor of *k*. However, it turns out that by organizing the neighbors in a special structured way (which together with a greedy routing scheme is known as Plaxton Routing), one can arrive at the destination in O(log n) number of hops while having each node retain only O(log n) number of neighbors.

In this structure, each node keeps track of its adjacent neighbors (the immediate nodes clockwise and counter-clockwise) to ensure correctness. For this assignment, we will abstractly call this the node's neighbor set (each DHT implementation has its own terminology). In addition to tracking its neighbor set, each node also tracks nodes in other parts of the ring that can serve as "short-cuts". These shortcuts are selected in a distribution that provides each node with sufficient out-pointers to a diverse set of points along the ring, while providing near authoritative knowledge of the nodes nearby. More formally, the ring is broken up into O(log m) regions, where m is the size of the ring i.e. $2^{128}$ for 128-bit hashes such as MD5. The first region encompasses half of the ring (or we can substitute half with a different constant factor); specifically the half encompassing IDs with a different most significant bit than the node's most significant bit. The second region is half the size of the first region encompassing IDs with the same most significant bit but with a different second most significant bit as the node. This follows for all remaining *log m* regions. Each node is responsible for keeping track of a constant number of nodes for each region. For this assignment, we will call this structure the node's routing table. The figure below illustrates the different region sizes for a node with ID of 0.



On receiving a get/put request, a DHT node determines the node in the union of its neighbor set and routing table that is the closest, based on the chosen distance metric, to the hash of the key. It will then forward the request to this node, which then performs the same computation based on its own neighbor set and routing table. This protocol ensures that each routing hop at least halves the remaining distance to the key's point in the ring. It is therefore easy to see that navigating the ring takes O(log m) hops. In fact, navigating the ring only requires O(log n) with high probability. The proof sketch is as follows: after *log n* hops and given that nodes are uniformly distributed, the distance between the current node's ID and the hash of the key is $2^m$ / n. The expected number of node IDs within a key range of $2^m$ / n is 1 and it is O(log n) with high probability. Therefore, even if we have to traverse all of these remaining nodes, the total number of hops to resolve this get/put query is still O(log n) with high probability.

## WatDHT Implementation

Now that you understand the theory behind DHTs, we can begin looking at the design of WatDHT. This DHT is mostly based on the Microsoft/Rice Pastry design, with some variations mainly to simplify the implementation and provide stronger consistency guarantees. The following are some key design parameters in WatDHT:

- Uses MD5 as the hash function, which dictates that the key space is $2^{128}$ in size.
- Key/values are stored in the closest counter-clockwise node in the ring.
- RPC is used to communicate between nodes.
- The routing table keeps track of one node for each of the first *(log m) / 32* regions rather than the full *log m* regions. Most DHTs only keeps track of a fraction of the regions as most of the later regions contain 0 nodes. This routing table size is especially small in order to demonstrate multi-hop routes even with a relatively small deployment.
- The neighbor set keeps track of the two closest clockwise nodes and two closest counter clockwise nodes. Note: the closest clockwise node is called a node's *successor* and the closest counter-clockwise is called a node's *predecessor*. Tracking the two closest nodes rather than just the closest node enables fast, local repairs of the ring for single node failures.

In order to allow interoperability between different WatDHT implementations, we will standardize on Apache Thrift (version 0.7) as the RPC implementation. Apache Thrift is a modern RPC implementation that is in heavy use within Facebook. It provides an IDL that is fairly flexible and intuitive to use. This assignment provides a WatDHT Thrift specification that defines the interface that your group will need to implement. In addition to implementing the functions defined in the Thrift specification, your program should also have the capability to perform periodic maintenance of the nodes in the neighbor set and the routing table. Finally, a delicate node join/migrate protocol must be followed in order to provide strong consistency. Let's look at each of the interface definitions in turn:

```
binary get(1: binary key) throws (1: WatDHTException err)

void put(1: binary key, 2: binary val, 3: i32 duration)
```

These are the two primary functions of WatDHT: *get* and *put*. They provide the interface to fetch the value of a given key and to insert a key/value pair. The duration parameter in the *put* function is the duration in seconds that the key/value pair is valid for. A negative value means it has an infinite duration, and a zero is equivalent to deleting the existing key/value pair. The following describes the basic implementation:

1. Check if the current node is the owner of this key/value pair. If so, perform get/put on the local data-structure. However, if this node has not successfully finished its initial *migrate_kv* call (see *migrate_kv* below), then both get/put should block until the *migrate_kv* call successfully completes.
2. Check if any node in the neighbor set is the owner of this key/value pair. If so, forward the request to the owner.
3. Forward the request to the closest node in the union of the routing table and neighbor set.

If a get request is received for a key that does not exist in the system, a WatDHTException is thrown back to the client (which may be several hops away) with the KEY_NOT_FOUND error code.

```
list<NodeID> join(1: NodeID nid)
```

A node sends a *join* request when it initially joins the DHT. The *nid* parameter is the NodeID, consisting of the ID, IP address and listening port of the joining node. The request is forwarded, possibly over several hops, to what will become the predecessor of the joining node. The receiving node returns its own NodeID and its neighbor set to the caller. The nodes on the return path will also add their own NodeID to the returning list. These nodes in the returning list are used to bootstrap the neighbor set and routing table of the joining node. All of the nodes on the path can use *nid* to update its neighbor set and/or routing table.

```
map<binary, binary> migrate_kv(1: binary nid) throws (1: WatDHTException err)
```

After a node joins the ring, it must migrate some of the keys/values from its predecessor to itself; specifically, those with key hashes that are bigger than or equal to its own ID. The *nid* parameter is the ID of the newly joined node, and the RPC request is sent directly to the newly joined node's predecessor. On receiving this request, the receiving node will first check if the calling node is its successor. If it is, then migrate to the caller all of its local key/value pairs with key hashes that are bigger than or equal to *nid*. If not, send back a WatDHTException with the INCORRECT_MIGRATION_SOURCE error code and set the *node* field to the receiving node's actual successor. The caller updates its neighbor set with this new node and resends *migrate_kv* to its new predecessor. If a node receives a

*migrate_kv* request before completing its own *migrate_kv* call, it will send back a WatDHTException with the OL_MIGRATION_IN_PROGRESS error code. The caller is responsible for calling *migrate_kv* again after a short timeout; the duration of the timeout increases exponentially with each OL_MIGRATION_IN_PROGRESS error.

**NOTE: The Winter 2012 version of *migrate_kv* unfortunately does not include the durations of the key/value pairs in the return value. Please work around this oversight by treating all migrated key/value pairs as having durations of -1.**

```
list<NodeID> maintain(1: binary id, 2: NodeID nid)
```

This function is for maintaining a node's routing table by finding nodes with specific IDs. Route this request to the node responsible for *id*, which should then return its own NodeID and its neighbor set. The parameter *nid* is the NodeID of the original caller; all of the nodes on the path can use this parameter to update its neighbor set or routing table.

```
list<NodeID> gossip_neighbors(1: NodeID nid, 2: list<NodeID> neighbors)
```

This function is for maintaining a node's neighbor set. The parameters *nid* is the ID of the caller, and *neighbors* is the neighbor set of the caller. It is sent directly to one of the caller's current neighbors. On receiving this request, the receiving node returns its own NodeID and its neighbor set, and uses *nid* and *neighbors* to maintain its own neighbor set.

```
binary ping()
```

This function is for checking if a node is still alive. On receiving this request, the receiving node returns its ID to the caller.

```
NodeID closest_node_cr(1: binary id)

NodeID closest_node_ccr(1: binary id)
```

This pair of functions is for maintaining the neighbor set when both neighbors in a particular direction are not available. For example, if both clockwise neighbors have left the system, the node sends a *closest_node_cr* request to its closest available clockwise node using its own ID as the call parameter. On receiving this request, the receiving node forwards the request to the closest clockwise node to *id* from the union of its neighbor set and routing table, or returns its own NodeID if it is the closest. The caller uses the return value to repair holes in its neighbor set.

**Node Startup and Maintenance**

On startup, a node takes as parameters its IP address, listening port and ID (its actual ID is the MD5 of this parameter), and, if it is the first node in the system, it is ready to receive requests from clients and other DHT nodes. Subsequent joining nodes are given the same parameters (with different values of course) plus the NodeID of an existing DHT node for bootstrapping. These new nodes must each perform the following operations:

1. Issue a *join* request to its bootstrap node to populate its neighbor set and routing table. Enable receiving requests once *join* completes, with the exceptions stated previously.
2. Issue *migrate_kv* to its predecessor.
3. Call *gossip_neighbors* to each of the nodes in its neighbor set.
4. Ensure that all regions in its routing table have an entry. For each region without an entry in the routing table, issue a *maintain* request for the last point in that region.

Additionally, each node must periodically check whether nodes in its neighbor set and its routing table are still available using *gossip_neighbors* for nodes in its neighbor set, and *ping* for nodes in its routing table. The period for *gossip_neighbors* and *ping* should be parameters that can be set independently. On discovering a node in its neighbor set is no longer available, a *gossip_neighbors* is called on the remaining available neighbor in the same direction. If both neighbors in a direction are unavailable, issue a *closest_node_cr* or *closest_node_ccr* to fix the neighbor set. Issue a *maintain* request, in the same way it was used at startup, to fix holes in the routing table.

## Submission Instructions

We will be using the "submit" system to handle submissions for this assignment. You can get more information on the submit tool using "man submit" or "submit -help". Please create a tarball named ${user-id-1}[-${user-id-2}].tar.gz (for example, ab123-cd456.tar.gz for students ab123 and cd456, or just ab123.tar.gz if he/she is working alone) that includes all of your source files, a Makefile that builds your system, and a README that includes the names and student IDs of the members in your group and details all of the external libraries you are using (excluding Thrift, Boost, and STL) with step-by-step instructions on how to install these libraries on the linux.student.cs.uwaterloo.ca machines. You can submit your assignment by issuing: "submit cs454 2 ab123-cd456.tar.gz". The assignment must be submitted before 11:59 PM March ~~12~~ 15, 2012 to avoid late penalties.

To test your system, we will run your program with the following commands:

For the first node in the system:

```
./server node_id ip port
```

For example: `./server 1 127.0.0.1 1234`

And for subsequent nodes:

```
./server node_id ip port ip_of_existing_node port_of_existing_node
```

For example: `./server 2 127.0.0.1 1235 127.0.0.1 1234`

Please note that these are example parameters. We will test your DHT across multiple physical nodes. If your main executable is not called "server" or do not take these parameters, please create a shell script called "server" that performs the necessary translation.

To simplify testing, please also set your neighbor set maintenance period to **10 seconds** and your routing table maintenance period to **30 seconds** (Note that these values are smaller than values that we might have said earlier. This change is mainly to expedite testing.)

The following is a list of simple sanity test cases you should consider when testing your system:

1. Node joins create a consistent ring. In steady state, if we issue a *gossip_neighbors* to nodes one by one along the ring, the returning results should be consistent (e.g. a node is its successor's predecessor), and we should be able to reach every node in the system.
2. Basic get/put requests work. We should be able to retrieve results that we inserted previously. Results do eventually expire if given a positive duration (WINTER 2012: assuming the key/value pairs have not been migrated). Delete works by performing a put with zero duration.
3. Node failures should not cause the ring to collapse. A node that loses both of its neighbors in a particular direction can correctly update its neighbor set by using *closest_node_cr* or *closest_node_ccr*. A simple test would be to kill two adjacent nodes simultaneously, wait for 10+ seconds, and check if the ring is still consistent.
4. Key/value pairs should always be stored in the closest counter-clockwise node in the ring. Be especially careful with handling concurrent joins. One easy way for us to test that key/value pairs are at the right locations is by creating, for each of your nodes, a test node with an ID that is one larger than your node. The *migrate_kv* call should then migrate every key/value pair (unless there happens to be a key that hashes to the same value as your node) from your node to our test node. We can then inspect the migrated items to see if they were in the right location.
5. Black box routing table tests are more difficult, and will likely involve introducing your node into our test DHT environment. We can then determine whether or not your node forwards requests to monotonically smaller regions and that it does not trivialize routing by keeping a list of every node in the system.

We will also inspect your source code. If we run into any trouble with compiling/testing your code, we will contact your group and will likely ask you and your group members to come in during office hours to remedy any issues. We will also contact your group if your submission fails a large number of automated tests. This will give an opportunity for your group to explain why your system is failing the tests, and to fix minor issues at our discretion.

**NOTE: Please check the course website for updates on this assignment. As this assignment is new, additional clarifications and minor changes to the requirements are inevitable; I will try to make this as painless as possible. Please feel free to contact me (bernard@uwaterloo.ca) or the TAs if you have any questions about the assignment.**