Project 2: Tapestry

Due: 11:59 PM, Feb 26, 2015

Contents

1 Introduction

The final project for CS138, Puddle Store, uses an underlying distributed object location and retrieval system (DOLR[1]), called Tapestry, to store and locate objects. This distributed system is similar to Chord in that it provides an interface for storing and retrieving key-value pairs. From an application's perspective, the difference between Chord and Tapestry is that in Tapestry the application chooses where to store data, rather than allowing the system to choose a node to store the object at. The application only publishes a reference to the object.

Tapestry is a decentralized distributed system. It is an overlay network that implements simple key-based routing. Each node serves as both an object store and a router that applications can contact to obtain objects. In a Tapestry network, objects are "published" at nodes, and once an object has been successfully published, it is possible for any other node in the network to find the location at which that object is published.

Node and Object Identifiers

Much like in other distributed systems, nodes and objects in the Tapestry network are each assigned their own globally unique identifier. In Tapestry, an ID is a sequence of n base-16 digits. An example of an ID is `2af5`.

Root Nodes and Surrogate Routing

In order to make it possible for any node in the network to find the location of an object, a single node is appointed as the "root" node for that object. The root node stores the list of locations of the object. Because Tapestry is decentralized and no single node has a global perspective on

[1] `http://en.wikipedia.org/wiki/Decentralized_object_location_and_routing`

the network, the root node for an object must be chosen in a globally consistent and deterministic fashion. The simplest choice of root node is the one which shares the same hash value as the object. However, it is common for there to be fewer nodes in the network than possible values in the space of hash values. For this reason, a "surrogate" node for an object is chosen to be the one with a hash value that shares as many prefix digits in the object's hash value as possible. Specifically, two hash values share a prefix of length n if, from left to right, n sequential digits starting from the leftmost digit are the same. For instance, in a network with nodes `1a9c`, `28ac`, `2d39`, and `ae4f`, the surrogate node for an object with the hash `280c` is `28ac` and the hashes share a prefix of length two. However, given this definition, the choice of surrogate node (from the same set of nodes as is in the previous example) would be ill-defined for an object with the hash `2c4f` because it shares a prefix of length one with both `28ac` and `2d39`. Therefore, we need a more general way of choosing the surrogate node when a single match is unavailable. Starting at the leftmost digit d, we take the set of nodes that have d as the leftmost digit of their hashes as well. If no such set of nodes exists, it is necessary to deterministically choose another set. To do this, we can try to find a set of nodes that share the digit $d + 1$ as their leftmost hash digit. Until a non-empty set of nodes is found, the value of the digit we are searching with increases (modulo the base of the hash-value). Once the set has been found the same logic can be applied for the next digit in the hash, choosing from the set of nodes we identified with the previous digit. When this algorithm has been applied for every digit, only one node will be left and that node is the surrogate.

To clarify, suppose a Tapestry network contains only the nodes `583f`, `70d1`, `70f5`, and `70fa`. The table below lists hypothetical object hashes and their corresponding surrogate nodes within this network.

Object Hash	3f8a	520c	58ff	70c3	60f4	70a2	6395	683f	63e5	63e9	beef
Surrogate Node	583f	583f	583f	70d1	70f5	70d1	70d1	70d1	70f5	70fa	583f

Tapestry Routing Tables

In order to allow nodes to locate objects stored at other nodes, each node maintains a routing table that stores references to a subset of the nodes in the network. A routing table has several *levels*; one level for each digit of the node's ID. In a Tapestry mesh that uses 40-digit IDs, the routing table therefore would have 40 levels. A node on level n of the routing table shares a prefix of length n with the local node.

Each level of the table consists of several *slots*; one for each unique digit. In a tapestry mesh that uses base-16 digits, each level of the routing table would therefore have 16 slots. A node in the d'th slot of the n'th has d as its n'th digit.

An example routing table for a node with the hash `3f93` is shown below.

Level	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0		1c42	2fe4	3f93	437e	5c2a	65bb	705b	8887	93cb			c3ca	d340	e9ce	f0d7
1	309c						362d						3c6f			3f93
2										3f93						
3				3f93												

The node stored at each entry in the routing table is the closest one that the local node is aware of. In a production implementation, distance between nodes is measured by the network latency between them, but for this project, distance is defined as the absolute value of the difference between hashes.

For robustness and redundancy, each slot of the routing table actually stores multiple references, typically 3.

Tapestry Back-pointer Tables

For additional connectivity, each node also stores *back-pointers* along with its routing table. Back-pointers are references to every node in the network which refers to the local node in their routing tables. These will become useful in maintaining routing tables in a dynamic network. When the local node adds or removes a remote node from its routing table, it notifies the remote node, who will then add a back-pointer to its back-pointer table.

Publishing and Retrieving Objects

Whenever an object is "published" by a node, that node routes towards the root node for the key, then registers itself on that node as a location of the key. Multiple nodes can publish the same object. A tapestry client wishing to lookup the object will first route to the root node of the object. The root node then informs the client of which Tapestry nodes are the ones that have published the object. The client then directly contacts one or more of those publishers to retrieve the actual object data.

Prefix Routing

The routing table at any given node does not store a reference to every other node in the network. Therefore, in order to find the root node for a particular ID, several nodes may be traversed until one is found that can definitively identify itself as the root node. The search for a root node may begin anywhere. Using the same logic that is used to choose a root node globally from the network, a node that matches some number of digits from the object's hash may be chosen from the routing table. In turn, the selected node's routing table is inspected and the next node in the route to the surrogate is chosen. At each successive node in the route, the number of digits that match the object's hash value increases until the last digit has been matched and the surrogate node has been reached. This type of routing is called "prefix routing", and the maximum number of hops required to reach the destination node is equal to the number of digits required to represent a hash value.

```
FindRoot(start, id)
    next = start
    current = nil
    do
        current = next
        next = current.getNextHop(id)
    while next is not null
    return current
```

In the version of Tapestry presented in the paper that introduced the system, when the location of an object is published to the object's surrogate node, the nodes encountered along the path to the surrogate node also have the location information for that object cached at them. This allows object lookups to finish in fewer hops from many starting locations in the network. Your implementation

is not required to have this feature, but it might be the starting point for an A-level extension to the final project.

Adding Nodes

To accommodate an increased workload, it is possible to add nodes to a Tapestry network. To perform this operation, the new node is assigned its ID and then routes towards the root node for that id. The root node initiates the transfer of all keys that should now be stored on the new node. The new node then iteratively traverses backpointers, starting from the root node, to populate its own routing table.

Acknowledged Multicast In Chord, when a new node joined the network, objects would be transferred from only one other node. Similarly in Tapestry, when a new node joins the network, other nodes transfer object references to it. Unlike Chord, it is possible for multiple different nodes to be storing references that should now be transferred to the new node. For example, suppose our Tapestry currently has nodes `a23b`, `285b` and `289a`, and our new node is `221f`. Then the root for the new node is `285b`. However, both `285b` and `289a` could be storing references that should be transferred to the new node. For example, `225f` would be stored on `285b`, and object `229f` would be stored on `289a`.

In general, if the new node has a shared prefix of length n with its root, then any other node that also has a shared prefix of length n with the new node could have relevant references. Such nodes are called *need-to-know* nodes.

To deal with this, the root node performs an *acknowledged multicast* when it is contacted by the new node. The multicast eventually returns the full set of need-to-know nodes from the Tapestry. The multicast is a recursive call - the root node contacts all nodes on levels $\geq n$ of its routing table; those nodes contact all nodes on levels $\geq n + 1$ of their routing tables; and so on. A node that is contacted during the multicast will initiate background transfer of relevant object references to the new node, trigger multicast to the next level of its routing table, then merge and return the resulting lists of nodes.

```
AddNodeMulticast(newnode, level)
  targets = routingtable.get(level)
  results = []
  for target in targets
    results.append(target.AddNodeMulticast(newnode, level+1))
  transferRelevantObjects(newnode)
  return results ++ targets
```

Backpointer Traversal Once the multicast has completed, the root node returns the list of need-to-know nodes to the new node. The new node uses this list as an initial *neighbour set* to populate its routing table. The node iteratively contacts the nodes, asking for their backpointers.

```
TraverseBackpointers(neighbours, level)
  for neighbour in neighbours
    AddToRoutingTable(neighbour)

  if level >= 0
    nextNeighbours = []
    for neighbour in neighbours
      nextNeighbours.append(neighbour.GetBackpointers(level-1))
    TraverseBackpointers(nextNeighbours, level-1)
```

Graceful Exit

Tapestry is extremely fault tolerant, so a node could leave without notifying any other nodes. However, a node can gracefully exit the Tapestry too. When a node gracefully exits, it notifies all of the nodes in its backpointer table of the leave. As part of this notification, it consults its own routing table to find a suitable replacement for the other node's routing table.

Fault Tolerance

The Tapestry network is designed to be extremely fault tolerant. As with any distributed system, some nodes may become unavailable unexpectedly. The mechanisms described in this section ensure that there is no single point of failure in the system. **Note that unlike with Chord you are expected to cleanly handle errors in this project, including the sudden crashing of nodes without cleanup.** In this project any time you make a remote method call you must check if an error is returned and handle the error appropriately.

Errors While Routing

When routing towards a surrogate node, it is possible that a communication failure with any of the intermediate nodes could impede the search. For this reason, routing tables store lists of nodes rather than a single node at each entry. If a failed node is encountered, the node that is searching can request that the failed node be removed from any routing tables it encounters, and resume its search at the last node it communicated with successfully (or if the last node it communicated with successfully is no longer responding it should communicate with the last successful node before that).

Loss of Root Node

Another potential loss from failure is the root node data. Two measures are taken to minimize the impact of failed root nodes. First, published objects continually republish themselves at regular intervals. This ensures that if a surrogate node goes down, a new surrogate node will eventually take its place. Unfortunately, there will still be a period of time in which the location information for the objects is unavailable. Applications built on top of Tapestry typically deal with this by storing each key multiple times with different salts, thereby offering backup locations when searching for an object. You do not have to implement salting in this assignment.

Loss of Replicas

Finally, applications built on top of Tapestry might wish to ensure that an object remains available at all times, even if the node that published it fails. In the *Publishing and Retrieving Objects* section, it was mentioned that multiple tapestry nodes can publish the same object. This means that client applications can learn of multiple locations of the object, so if the object becomes unavailable in one of these locations, the client can simply contact another of the nodes. On the root node for a key, when a long enough period of time elapses without receiving an object republish notification from a publishing node, then the object expires and is removed.

Miscellaneous

The cases listed above are the common issues which can arise due to network errors. There are other more obscure ways in which surrogates may become unreachable for a short time when nodes join or fail in a certain order. Tapestry's method for dealing with this is to assume that there are enough seeded hash values for a given object that not all seeds will become unreachable due to such errors, and those which do become unreachable will be corrected when the replica performs its periodic republishing.

2 The Assignment

A large amount of support code has been given to you for this assignment. All of the required data structures are implemented in the support code. The code you will write is related to routing in the network, storing and retrieving object location data, and coping with failures. Please become very familiar with all of the support code before beginning to implement any of the features. The comments for each method that you will fill in should give you a good idea of how to proceed. The code you must write is marked with `//TODO students` comments and is spread across the Go files in the `tapestry` directory. Feel free to add helper functions but you must implement the following functions:

- `id.go`
  - `func SharedPrefixLength(a ID, b ID) int`
  - `func (id ID) BetterChoice(first ID, second ID) bool`
  - `func (id ID) Closer(first ID, second ID) bool`
- `routingtable.go`
  - `func (t *RoutingTable) Add(node Node) (bool, *Node)`
  - `func (t *RoutingTable) Remove(node Node) bool`
  - `func (t *RoutingTable) GetLevel(level int) []Node`
  - `func (t *RoutingTable) GetNextHop(id ID) Node`
- `tapestry-localimpl.go`
  - Functions invoked by local clients

    * `func (local *TapestryNode) Join(otherNode Node) error`
    * `func (local *TapestryNode) Leave() error`
    * `func (local *TapestryNode) Publish(key string) (chan bool, error)`
    * `func (local *TapestryNode) Lookup(key string) ([]Node, error)`
  - Functions invoked over RPC by other Tapestry nodes
    * `func (local *TapestryNode) AddNodeMulticast(newnode Node, level int) ([]Node, erro`
    * `func (local *TapestryNode) NotifyLeave(from Node, replacement *Node) error`
    * `func (local *TapestryNode) GetNextHop(id ID) (bool, Node, error)`
    * `func (local *TapestryNode) Register(key string, replica Node) (bool, error)`
    * `func (local *TapestryNode) Fetch(key string) (bool, []Node, error)`
    * `func (local *TapestryNode) Transfer(from Node, replicamap map[string][]Node) error`
  - Utility Functions
    * `func (local *TapestryNode) addRoute(node Node) error`
    * `func (local *TapestryNode) findRoot(start Node, id ID) (Node, error)`

A partial implementation of `Join` in `tapestry-localimpl.go` is provided to demonstrate invocation of local and remote methods, and error handling.

Remote Procedure Call (RPC)

The `Tapestry` struct exposes an API for invoking methods on remote Tapestry nodes.

The `TapestryNode` struct maintains the state of the local tapestry node, such as the routing table, backpointers table, and object store. Each `TapestryNode` instance has a `Tapestry` member variable named `tapestry`.

We have handled the implementation of remote method invocation; your job is to provide the local method implementations. As part of your implementations, you will need to invoke some methods on remote tapestry nodes. For each `TapestryNode` method that is commented as being an RPC method, you will find a corresponding `Tapestry` method for remote invocations. For example:

```
func (local *TapestryNode) Fetch(key string) (bool, []Node, error)
```

has a corresponding method:

```
func (tapestry *Tapestry) fetch(remote Node, key string) (bool, []Node, error)
```

Notice that the `Tapestry` method takes one additional parameter: the node on which to invoke the method. Therefore, to invoke locally, you would call in your code:

```
local.Fetch("myKey")
```

To invoke this method on a remote node, you would call:

```
local.tapestry.fetch(remoteNode, "myKey")
```

3 Testing

We expect to see several good test cases. This is going to be worth a portion of your grade. Exhaustive Go tests are sufficient. You can check your test coverage by using Go's coverage tool[2].

A number of Tapestry constants are defined in `tapestry.go`. You can change these constants during development to simplify debugging. For your own unit tests, you may assume we will use the default values specified in `tapestry.go`. However, our own testing suite may use different values for these parameters, so do not hard-code values in your implementation.

a. **cli.go** - This is a Go program that serves as a console for interacting with Tapestry, creating nodes and querying state on the local node(s). We have kept the CLI simple but you are welcome to improve it as you see fit.

   Assuming your CS138 git repository is located at ~`/course/cs138`, you can build and run the CLI as follows

   ```
   $ cd ~/course/cs138/tapestry
   $ go build -o cli
   $ ./cli
   ```

   You can pass the following arguments to `cli.go`:

   - -p <int>: The port to start the server on. By default selects a random port.
   - -c "host:port": Address of an existing Tapestry node to connect to
   - -d=true: Enable or disable debug
   - -help: Print usage

   You get the following set of commands available to you in the terminal:

   - table
     - Print this node's routing table
   - backpointers
     - Print this node's backpointers
   - objects
     - Print the object replicas stored on this node
   - put <key> <value>
     - Stores the provided key-value pair on the local node and advertises the key to the tapestry
   - lookup <key>
     - Looks up the specified key in the tapestry and prints its location
   - get <key>
     - Looks up the specified key in the tapestry, then fetches the value from one of the returned replicas

[2] `http://blog.golang.org/cover`

- remove <key>
  - Remove the value stored locally for the provided key and stops advertising the key to the tapestry
- list
  - List the keys currently being advertised by the local node
- debug on|off
  - Turn debug messages on or off
- leave
  - Instructs the local node to gracefully leave the Tapestry
- kill
  - Leaves the tapestry without graceful exit
- exit
  - Quit the CLI

If you are confused about the behaviour of any of these commands, feel free to refer to the demo at `/course/cs138/pub/tapestry`.

b. You're encouraged but not required to write client applications, using the `Tapestry` struct and its provided methods, to test your implementation.

4 Getting Started

Before you get started, please make sure you have read over, understand, and have set up all the common code.

To get started, clone your and your partner's Github repository[3] to your CS 138 course directory. Once you have done that, you will want to merge in the Tapestry support code:

```
git remote add support git@github.com:brown-csci1380/projects.git
git pull support master
```

If you've never merged in Git before, or you want a refresher, Github has a guide[4] that you may find useful.

5 Handing in

You need to write a README, documenting any bugs in your code, any extra features you added, and anything else you think the TAs should know about your project. Once you have completed your README and project, you should hand in your `tapestry` by running

```
/course/cs138/bin/cs138_handin tapestry
```

[3] `https://github.com/brown-csci1380`

[4] `https://help.github.com/articles/resolving-a-merge-conflict-from-the-command-line/`

to deliver us a copy of your code.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS138 document by filling out the anonymous feedback form:

`http://cs.brown.edu/courses/cs138/s15/feedback.html`.