Project 2: Tapestry

Due: 11:59 PM, Feb 28, 2019

Contents

1 Introduction

The final project for CS1380, PuddleStore, 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.

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.

2 Tapestry Protocol

2.1 Identifying Nodes and Objects

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 fixed-length sequence of base-16 digits.

2.2 Root Nodes

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 reference to the node that actually stores the object.

Because Tapestry is decentralized, and no single node has a global perspective on 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.

2.2.1 Selecting Root Nodes

For this reason, a "root" node for an object is chosen to be the one with a hash value that shares the most prefix digits with the object's hash value.

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

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 example, in a network with nodes `1a9c`, `28ac`, `2d39`, and `ae4f`, the root node for an object with the hash `280c` is `28ac` and the hashes share a prefix of length 2, because the other nodes share a prefix of length 1 or 0. However, given this definition, the choice of root 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 root node when a single match is unavailable.

Starting with the value v of the leftmost digit, we take the set of nodes that have this value 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 value $v + 1$ as their hash's leftmost value. 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 a 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 root.

2.2.2 Example: A Tapestry Network's Objects and their Roots

To clarify, suppose a Tapestry network contains only the nodes `583f`, `70d1`, `70f5`, and `70fa`.

To find the root node for an object with a hash of `60f4`, we first consider the leftmost digit's value, 6. None of the network nodes share this leftmost value, so we check if any network nodes have the leftmost value $6 + 1 = 7$. `70d1`, `70f5` and `70fa` do, so we take this set and go to the next digit. The object hash's next digit, 0, is shared with all the network nodes in the current set, so we go to the next digit. The third digit of the object's hash, f, is shared with only `70f5` and `70fa` so we take this smaller set and go to the last digit. The object's hash has a final digit of 4, which doesn't match either 5 or a, so we try with $4 + 1 = 5$. This matches the network node with a hash of `70f5`, so this node is the object's root node. If the object's hash had been `60f6`, its root node would be the network node with the hash `70fa`.

The table below lists hypothetical object hashes and their corresponding root nodes within this network.

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

2.3 Tapestry Node State

Some state is maintained on each Tapestry node to carry out its ability to route to nodes and lookup objects.

2.3.1 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 would thus have 40 levels. The level represents the size of the shared prefix with the local node; that is, 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} level has d as its n^{th} digit (keep in mind that n is zero-indexed!). For example, in the table given, the entry at level 1 in slot 6 (`362d`) shares a prefix of length 1 (because it's on level 1) and has 6 as its first digit (because it's in slot 6). If there had been an ID of `3782`, then on level 1 in slot 7 we would see this ID.

In summary, a routing table entry is defined by two numbers: its level and slot. The level represents the length of the shared prefix with the local node, and the slot represents the first digit of the remote node after the shared prefix.

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	a	b	c	d	e	f
0		1c42	2fe4	3f93	437e	5c2a	65bb	705b	8887	93cb			c3ca	d340	e9ce	f0d7
1	309c						362d						3c6f			3f93
2										3f93						
3				3f93												

If the local node knows about more than one node that fits into a cell, the one that is stored at each entry in the routing table is the closest one to the local node. In a production implementation, distance between nodes is measured by the network latency between them, but for this project, we arbitrarily define distance as the absolute value of the difference between hashes.

In addition, for robustness and redundancy, each slot of the routing table actually stores multiple references, typically three. The first one is the closest node, and the others are backups in case the first one fails to respond to requests. These are sorted by distance to the local node.

2.3.2 Backpointer Tables

For additional connectivity, each node also stores *backpointers* in addition to its routing table. Backpointers are references to every node in the network which refers to the local node in their own 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 update its backpointer table.

2.4 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 root 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 root 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, when the location of an object is published to the object's root node, the nodes encountered along the path to the root 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 PuddleStore.

2.5 Publishing and Retrieving Objects

When 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.

2.6 Adding Tapestry 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.

2.6.1 Acknowledged Multicast

In Tapestry, when a new node joins the network, other nodes transfer object *references* to it, i.e. it takes over and becomes the root for objects whose IDs now closely match its ID. 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`. The root for the new node is thus `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 the current root for its ID, 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 a background transfer of relevant object references to the new node, trigger a multicast to the next level of its routing table, then merge and return the resulting lists of nodes (removing duplicates) while adding the new node to its routing table.

```
AddNodeMulticast(newnode, level)
  targets = routingtable.get(level) // Include local node
  results = []
  for target in targets
            results.append(target.AddNodeMulticast(newnode, level + 1))
  self.addRoute(newnode)
  transferRelevantObjects(newnode)

  return merge(results, targets)
```

2.6.2 Backpointer Traversal

Backpointer traversal is used to find the best/closest set of nodes to fill the routing table with. This algorithm is different from the one in lecture, but as we already require you to use backpointers for the graceful exit, we also require you to use the backpointer based algorithm to fill the routing table while adding a node.

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 *neighbor set* to populate its routing table. The node iteratively contacts the need-to-know nodes, asking for their backpointers. Once the node has compiled backpointers from each of its need-to-know nodes, it is necessary to remove duplicate nodes, and trim the list of nodes to visit to K, as the number of nodes we search can get very large, but we only care about the closest few nodes. We give you the constant K in `node_init.go`.

```
TraverseBackpointers(neighbors, level)
  if level >= 0
    nextNeighbors = neighbors

    for neighbor in neighbors:
        nextNeighbours.append(neighbor.GetBackpointers(level))

    AddAllToRoutingTable(nextNeighbors)
    removeDuplicatesAndTrimToK(nextNeighbors)
    TraverseBackpointers(nextNeighbors, level-1)
```

2.7 Graceful Exits

A good implementation of 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 routing tables of all the other nodes.

Objects stored at exiting nodes will be lost and no objects are transferred.

2.8 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. **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.

Note that when a node crashes, the objects stored at that node will be lost. However, it is the responsibility of the client application that uses the Tapestry network to put duplicate objects across the network. You don't need to worry about preventing data loss in this case.

2.8.1 Errors While Routing

When routing towards a root 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 slot. 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. If the last node it communicated with successfully is no longer responding, it should communicate with the last successful node before that.

2.8.2 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 root node goes down, a new root node will eventually take its place. Unfortunately, there will still be a period of time in which the location information for these objects is unavailable.

Second, applications built on top of Tapestry typically store 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.

2.8.3 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 nodes storing a particular object. Thus, if the object becomes unavailable at one of these nodes, the client can simply contact another one of the nodes. On the root node for a key, when a long enough period of time has elapsed without receiving an object republish notification from a publishing node, the object expires and is removed.

2.8.4 Miscellaneous

The cases listed above are the common issues which can arise due to network errors. There are other more obscure ways in which roots 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 salted hash values for a given object that not all salts will become unreachable due to such errors, and those which do become unreachable will be corrected when the replica performs its periodic republishing.

3 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.

3.1 Function Stubs

The code you must write is marked with `// TODO students should implement this` comments and is spread across the Go files in the `tapestry` directory. Feel free to add helper functions. You must implement the following functions:

- `id.go`

```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
```

- `routing_table.go`

```go
func (t *RoutingTable) Add(node RemoteNode) (added bool, previous *RemoteNode)
func (t *RoutingTable) Remove(node RemoteNode) (wasRemoved bool)
func (t *RoutingTable) GetLevel(level int) (nodes []RemodeNode)
```

```
func (t *RoutingTable) GetNextHop(id ID) (node RemoteNode)
```

- `node_init.go`

  Functions for creating a Tapestry node and joining an existing network

```
func (local *Node) Join(otherNode RemoteNode) (err error)
func (local *Node) AddNodeMulticast(newnode RemoteNode, level int)
                    (neighbors []RemoteNode, err error)
func (local *Node) addRoute(node RemoteNode) (err error)
```

- `node_core.go`

  Functions for publishing and looking up objects in the network

```
func (local *Node) Publish(key string) (done chan bool, err error)
func (local *Node) Lookup(key string) (nodes []RemoteNode, err error)
func (local *Node) GetNextHop(id ID)
                    (morehops bool, nexthop RemoteNode, err error)
func (local *Node) Register(key string, replica RemoteNode)
                    (isRoot bool, err error)
func (local *Node) Fetch(key string)
                    (isRoot bool, replicas []RemoteNode, err error)
func (local *Node) Transfer(from RemoteNode, replicamap map[string][]RemoteNode)
                    (err error)
func (local *Node) findRoot(start RemoteNode, id ID) (RemoteNode, error)
```

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

3.2 Remote Procedure Call (RPC)

RPC is a technique that allows programs to call procedures on other machines. When one machine calls a procedure on another machine using RPC, the execution is suspended on the first machine until the call on the second machine returns and its return value is received by the original machine. In the stencil code, `tapestry_rpc_client.go` contains the functions that handle calling procedures on other nodes. `tapestry_rpc_server.go` contains the local implementations of the functions being called on a machine.

RPCs for Tapestry are handled by the gRPC library which runs on top of Protocol Buffers, a way of generating communication files from a `.proto` file. You will find this code pre-generated for you in `tapestry_rpc.pb.go`, and the source file it was generated from in `tapestry_rpc.proto`. In future projects, you will be asked to do more of this yourself, so it is worth taking a glance at both these files.

We divide RPCs into two categories, client functions and server functions. Server functions expose local methods to other nodes, and are listed in `tapestry_rpc_server.go`. Client functions handle connection to a remote node and calling a function on it, and are listed in `tapestry_rpc_client.go`. To expose local functions as RPCs, your `TapestryNode` object needs to implement the TapestryRPCServer interface generated by gRPC. These methods follow a very particular signature:

```
func (local *TapestryNode) XxxCaller(ctx context.Context, req *Request) (*Reply, error)
```

We've adopted a convention of using the suffix "Caller" to differentiate these methods from the other methods of `TapestryNode`. You are responsible for implementing around half of them in `tapestry_rpc_server.go`, but they have all been outlined for you there. Each of these Caller functions acts on the local node after receiving a request from a remote node. So each needs to:

1. Unpack the arguments from its request struct.
2. Call the corresponding local method.
3. Pack the results into a reply struct.
4. Return the reply struct and any error.

Client functions are methods of the `RemoteNode` struct, and handle invoking a method over a network connection. We use the "RPC" suffix to denote these methods, and you will also be implementing about half of them in `tapestry_rpc_client.go`. Each client function needs to do the following:

1. Pack its arguments into the appropriate request struct.
2. Obtain a network connection to the remote node.
3. Invoke the method over the network connection, and receive a reply struct and an error.
4. Unpack the reply struct, return these values and any error and if there was an error, close the network connection.

We've given you a `ClientConn` method of `RemoteNode` that will establish or reuse a connection to a remote node, and return a `TapestryRPCClient` that will let you call the "Caller" functions, as well as several other utility functions in `tapestry_rpc_client.go` and `tapestry_rpc_server.go` to convert between message types and to error check RPCs. Feel free to use these in your implementations, and to copy the patterns from the other RPC client and server functions.

These `XxxRPC` and `XxxCaller` functions shouldn't contain any application logic inside them; all they should need to handle is unpacking arguments and passing them to a different function. For instance: one node invokes `AddNodeRPC`, which obtains a client connection and calls `AddNodeCaller`. On the remote node, a new Go routine begins `AddNodeCaller`, which calls `AddNode` on its local node. In general, `<function>RPC` calls `<function>Caller`, which calls `<function>`.

3.3 A Note About Context

All of the functions generated by gRPC take a `context` parameter, which we aren't using for this project. Feel free to use `context.Background()` whenever you need to provide one.

4 Demo

A TA implementation of Tapestry is available at

```
/course/cs1380/pub/tapestry/{linux,darwin,windows}/tapestry
```

Once your implementation is sufficiently functional, you should test with the TA implementation for interoperability.

5 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 `node_init.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 `node_init.go`. However, our own testing suite may use different values for these parameters, so do not hard-code values in your implementation.

When writing your unit tests, you may run into an error from gRPC along the lines of `socket: too many open files`. Each network connection your computer maintains uses up a file descriptor, as if you had opened a file for reading or writing, and there's a limit to how many you are allowed to have open at once. On macOS, this limit is relatively low, at 256 per terminal window. If your tests give you the `Too many open files` error, try increasing the limit with

```
ulimit -n <amount>
```

and document this need in your README.

- `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. We have kept the CLI simple but you are welcome to improve it as you see fit.

  You can build and run the CLI as follows:

  ```
  $ cd $GOPATH/src/github.com/brown-csci1380/<your-repo>/tapestry
  $ go install
  $ tapestry
  ```

  Note: if running `tapestry` doesn't run the CLI, you can run `$GOPATH/bin/tapestry`.

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

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

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

You have the following set of commands built into the CLI:

- `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
- `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 behavior of any of these commands, feel free to refer to the demo at `/course/cs1380/pub/tapestry`.

You are encouraged, but not required, to write client applications (that is, applications that use your Tapestry implementation to store objects), using the `tapestry/client` package and its provided methods, to test your implementation.

6 X-Trace

In this project, we'll be providing support for a different logging framework, which is immune to some of the timing and race conditon issues you may have had with regular log statments and multiple goroutines, if instrumented successfully. X-Trace is a "logging framework for distributed systems", and is described in Chapter 3 of Rodrigo Fonseca's PhD thesis. X-Trace organizes your log statements by causal order, instead of absolute time, and generates traces that show network communication between nodes, and parallel processes on a single node. If you want to run it on Go, however, you need to make a small modification to the `runtime` package of your Go installation. We'll provide more details in a handout soon if you want to use X-Trace for this project. Projects that use X-Trace well can earn some extra credit.

7 Getting Started

Remember, if you write code on a department machine, you must use go1.11 instead of just go (ie go1.11 install or go1.11 test)

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

The stencil can be found at `http://cs.brown.edu/courses/csci1380/s19/content/projects/tapestry.zip`.

You should unzip this, and then place its contents in your gopath at src/github.com/brown-csci1380/<your-team-name>.

We highly encourage you to work in groups of two, but we understand that in some situations a group of three may be necessary. If you work in a group of three, you must either include X-Trace in your Tapestry (and do a good job of it!) or implement an additional feature. Additional features include publishing path caching, hotspot caching, hash salting, object re-replication, and erasure coding. Stop by TA hours to learn more about what these are! If you work in a group of three, you must contact the TAs and let them know you intend to work in a group of three, and if you will be using X-Trace or implementing additional features.

Working alone is not allowed for this project. If you do not have a partner for any reason, please attempt to find one thru the piazza partner search functionality, and if this is not successful please email the htas for assistance.

8 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. Note that this README should be located in the `tapestry` folder. Do not use the README created at the root of your repository. This README is in the same top-level folder as the folders for all of your CS 1380 projects so it does not clearly correspond to any one project.

Once you have completed your README and project, you should hand in your `tapestry` by running

```
/course/cs1380/bin/cs1380_handin tapestry
```

from the root of your repository 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 CS1380 document by filling out the anonymous feedback form:

`http://cs.brown.edu/courses/cs138/s19/feedback.html`.