

CS138 Programming Assignment 2: Tapestry

<i>Assignment Out:</i>	Feb. 10, 2009
<i>Helpsession:</i>	Feb. 12, 2009 (Location TBD, 7pm)
<i>Assignment Due:</i>	Mar. 2, 2009 (11:59 pm)

1 Tapestry

The final project for CS138, Puddle Store, uses an underlying distributed object location and retrieval system (DOLR), called Tapestry, to store and locate objects. This distributed system is similar to the one that you implemented in the first assignment 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 the application stores objects at specific nodes in the network, rather than allowing the system to choose a node to store the object at. Tapestry is a decentralized distributed system. 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.

Surrogate Nodes

Much like in other distributed systems, each node in the Tapestry network is assigned a globally unique integer that identifies it. Obtaining this integer is usually done by taking the hash value of the IP address or MAC address of the machine on which the node is running. For instance, a node with the IP address 128.148.33.143 might have the hash value 0a32. In this particular network, hash values are represented in hexadecimal form and they are stored as 16 bit values, and so the maximum possible hash value is FFFF.

Just as the nodes in the network are identified by a hash value, each object (or key-value pair) stored in the Tapestry network also has a hash value which identifies it. Unlike for nodes, it is not required that two distinct objects have different hash values. In order to make it possible for any node in the network to find the location of any object, a single node is appointed as the "surrogate" node for that object. The surrogate node stores information about where copies of an object are stored.

Because Tapestry is decentralized and no single node has a global perspective on the network, the surrogate node for an object must be chosen in a globally consistent and deterministic fashion. The simplest choice of surrogate 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, the surrogate node for an object is chosen to be the one with a hash value that shares as many of the digits in the object's hash value as possible. Specifically, two hash values share n digits 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 two digits. 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 one digit with both `28ac` and `2d39`. Therefore, we need a more general way of choosing the surrogate node when the perfect 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 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	<code>3f8a</code>	<code>520c</code>	<code>58ff</code>	<code>60f4</code>	<code>70a2</code>	<code>70f7</code>	<code>beef</code>
Surrogate Node	<code>583f</code>	<code>583f</code>	<code>583f</code>	<code>70f5</code>	<code>70d1</code>	<code>70fa</code>	<code>583f</code>

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. This routing table has several levels, each of which correspond to a digit of the node's hash value. Nodes with hashes that share no digits with hash value of the local node are stored at level 0 of the table. At level n , the hashes share n digits.

At each level of the table, the nodes are indexed by the first digit of their hash that does not match the hash of the local node. 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		<code>1c42</code>	<code>2fe4</code>	<code>3f93</code>	<code>437e</code>	<code>5c2a</code>	<code>65bb</code>	<code>705b</code>	<code>8887</code>	<code>93cb</code>			<code>c3ca</code>	<code>d340</code>	<code>e9ce</code>	<code>f0d7</code>
1	<code>309c</code>						<code>362d</code>						<code>3c6f</code>			<code>3f93</code>
2										<code>3f93</code>						
3				<code>3f93</code>												

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 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.

Publishing and Retrieving Objects

Whenever an object is “published” at a node, that node sends a message to the surrogate node for that object to inform the surrogate that a copy of the object has been stored locally. There may be many copies of an object, each at different nodes in the network, and references to each node are associated with the hash of the object at its surrogate node. When other nodes in the network attempt to retrieve that object, they send a message to its surrogate asking for the closest copy of the object.

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 surrogate for a particular hash, several nodes may be traversed until one is found that can definitively identify itself as the surrogate node. The search for a surrogate node may begin anywhere. Using the same logic that is used to choose a surrogate 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.

```
routeToSurrogate(G,h)
  N = G
  C = null
  do
    C = N
    N = C.nextHop(h)
  while N is not null
  return C
```

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 it 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 you may add it for extra credit.

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 hash value and then attempts to route towards to surrogate node for that hash value, beginning at any node. At each node it encounters along the way, the new node copies the level of the routing table corresponding to the number of hops it has made so far until it reaches the surrogate node. It then copies any relevant object location data from the surrogate and announces its presence in the network. The new node is first added to the routing table at its surrogate node and then floods the surrogate node's back-pointers with messages. The nodes receiving these messages will add the new node to their routing tables if appropriate. The new node will then visit all of the back-pointers at those nodes which it just contacted, and so on until all the nodes in the network are aware of the new node. Adding a nodes to the network is expected to be an infrequent operation, and thus the expensive nature of the operation will only impact the network occasionally.

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.

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.

Another failure that the system must protect against is that of the surrogate node. Should any node become unavailable, the object location information that it stored will also be unavailable. Two measures are taken to minimize the impact of failed surrogate nodes. First, replicas of objects republish their locations 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. For this reason, we assign multiple surrogate nodes to each object. Whenever the location of an object is published, its hash value is “salted” so that rather than having a single hash value, multiple hash values are associated with the object. In a large enough network, each hash value will map to a different surrogate node. When searching for object location information, each possible hash value for the object must be tried before determining that the object does not exist. Salting is performed by simply appending a number to the end of an object's key (for instance, the key `myobject` might become `myobject0`, `myobject1`, and `myobject2`).

Finally, the Tapestry network must ensure that an object remains available at all times, even if the node storing it fails. In the *Publishing and Retrieving Objects* section, it was mentioned that

multiple locations for the object can be stored at an object's surrogate node. Each of the nodes that have a copy of an object periodically republish their locations to the surrogate node. When a long enough period of time elapses and the surrogate node has not heard from an object replica, that location gets removed. It is up to the client application to choose and maintain its object replicas. Should a malfunctioning replica be contacted during the search for an object, that replica is removed at the surrogate node as well.

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.

In `TapestryNeighborMapLocal.java`, you should implement the following method:

```
public TapestryNodeCache nextHop(BigInteger hash)
```

In `TapestryNodeLocal.java`, you should implement these methods:

```
public boolean notifySurrogate(TapestryNodePrivate node)
public void hello(TapestryNodePrivate node)
private boolean republishObject(String key)
public Serializable findObject(String key)
public boolean publishLocation(String key, BigInteger hash, TapestryNodeCache cache)
public boolean getLocation(String key, BigInteger hash, TapestryResponse response)
public void removeLocation(String key, TapestryNodeCache cache)
private void removeHash(String key, BigInteger hash)
public void publishObject(String key, Serializable data)
public Serializable getObject(String key)
```

3 Testing

Given no failures, every node in your Tapestry network should be able to publish and retrieve objects. When new nodes enter the network, pre-existing objects should remain available, even if the surrogate nodes for the salted hashes of the object change. Unless every surrogate node or every node replicating an object fails, every object must remain accessible from every node in the network.

The fault tolerance features of Tapestry are designed with a large network in mind, so it is important that you test your implementation with a large number of nodes. Some bugs may not

become apparent until the network becomes large enough. Begin by writing enough code to publish and retrieve object from the network while add nodes on the fly. Once that functionality is stable, proceed by implementing the fault tolerance mechanisms and test your implementation on larger networks. Provided with the support code is an `ant` build file with `run` and `debug` targets which invoke the `main` method in the `TapestryDriver.java` file. Write your test code in this file. You must provide evidence that you have tested your code thoroughly. **A full 15% of your grade will be based on the quality of your test code.**

To aid you during the early stages of development, a console utility, located at

```
/course/cs138/demo/tapestry/console.sh
```

has been provided for you. It will allow you to access basic information about each node in the network, publish and retrieve objects, and inspect routing tables and object location information. It is not a replacement for thorough test code, however.

4 Extra Credit

There is ample room for additional features in this assignment. If you're interested in going beyond the basic implementation, ask the TAs about what you can add.

5 Code Exchange

To get started, copy the `/course/cs138/asgn/tapestry` directory to your CS138 course directory.

6 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. You should hand in `tapestry` by running `"/course/cs138/bin/cs138_handin tapestry"` from your project's directory.