CS 549—Fall 2024
Distributed Systems and Cloud Computing
Assignment Three—Events

In the previous assignment, you developed a simple peer-to-peer distributed hash table for sharing bindings of keys to values. The protocol for retrieving bindings from the database is pull-based: a client (node) searches the network for the bindings for a particular key. In this assignment, you will extend this with a push-based protocol: Clients will be notified when new bindings are added for a key. The implementation of this push protocol will be based on the server-streaming API in gRPC.

The command line interface (CLI) for a node should support three commands:

1. `listenOn <key>` records an interest in being notified of changes in bindings for the specified key. For this assignment, you can focus just on notifications of the addition of new bindings.
2. `listenOff <key>` removes an interest in being notified of changes in bindings for the specified key.
3. `listeners` lists all of the keys for which this node has registered listeners in the network.

When a listener is registered, the node searches the network for the node where bindings for the corresponding key are stored. A Web service call is made to that node, to open up a communication stream for that node to send notifications to the client node when a binding for that key is stored. At the client node, a listener callback is used to process event notifications from the server node.

When the user wants to turn off notifications for a particular key, the client must contact the server to cancel the notifications, and therefore clients must keep track of the servers for which they are listening for notifications. When a server notifies a client that the event stream is closing (using `onComplete`), the client can remove this record of the listener.

The change to the server interface from the previous assignment is the addition of two operations:

```
rpc listenOn (Subscription) returns (stream Event);

rpc listenOff (Subscription) returns (google.protobuf.Empty);
```

A subscription identifies the node that is requesting the stream of event notifications; it is used on the server side to identify the listener when they later request the stream of notifications to be closed.

```
message Subscription {
  int32 id = 1;  // Identifies the subscriber on the notifier
  string key = 2;
}
```

The stream of events that the server returns to the requester is made up messages of this form:

```
message Event {
  oneof bindingEvent {
    Binding newBinding = 1;
    google.protobuf.Empty movedBinding = 2;
  }
}
```

There are two types of events notifications:

1. Addition of a binding of an additional value to a key: The client simply displays this information on the console.
2. Movement of a binding to another node: This may happen because another node has joined the network and made itself the predecessor of the original "server" node, and one of the keys that the client is watching has moved to the new node. In this case, when notified, the client must stop listening at the original node and start listening at the new node, which it finds in the same way it found the original node.

Whereas so far you have only used blocking stubs to make calls from the client, the stream of event notifications must be requested using an asynchronous stub, which provides this as the type of operation for turning on listening:

```
public void listenOn(Subscription subscription,
                     StreamObserver<Event> responseObserver) { ... }
```

The `responseObserver` processes notifications to the client of new bindings for the key specified in the subscription. Rather than pollute the application code with the details of the gRPC event type, we define this API that is used internally on the client and stub:

```
public interface IEventListener {

    public void onNewBinding(String key, String value);

    public void onMovedBinding(String key);

    public void onClosed(String key);

    public void onError(String key, Throwable throwable);

}
```

On the client side, a listener callback with this interface is provided to the operation for requesting a stream of event notifications from the server.

```
// Client-side
public void listenOn(NodeInfo node, Subscription subscription,
                     IEventListener listener) throws DhtBase.Failed
```

An `EventConsumer` object wraps a listener of this `IEventListener` type and translates from gRPC notifications to the notifications expected by the `IEventListener` callback (so the `EventConsumer` callback is passed as the response observer to the client stub `listenOn` operation):

```
public class EventConsumer implements StreamObserver<Event> { ... }
```

On the server side, the `StreamObserver` object that is the second argument to the server stub `listenOn` operation is wrapped with an `EventProducer` object:

```
public class EventProducer implements IEventListener { ... }
```

This translates calls in the server on the `IEventListener` interface into the generation of events that are returned from the server to the client as a stream of responses:

```
// Server-side
public void listenOn(int id, String key, EventProducer eventProducer)
```

The API for the singleton state object is extended with these client-side operations for recording which nodes we are listening for event notifications from, and listing all keys for which we are listening:

```
public boolean startListening(String key, NodeInfo target);

public NodeInfo getListeningTarget(String key);

public void stopListening(String key);

public void listeningFor(OutputStream out);
```

This service-side API is used to keep track of the event streams for the clients that have requested notifications:

```
public interface IEventBroadcaster {

    public void addListener(int id, String key, EventProducer listener);

    public void removeListener(int id, String key);

    public void broadcastNewBinding(String key, String value);

    public void broadcastMovedBinding(String key, NodeInfo node);

}
```

Once you have your code running locally, you should test it on some EC2 instances that you define. Copy the server jar file to the EC2 instance using "`scp`," using the "`-i`" option to define the private key file that you use to authenticate yourself to the instances. Then ssh to the instance and run the server script. It should be in a directory of the form

```
/home/ec2-user/tmp/cs549/dht-test
```

that you should have created. Now you should be able to run network nodes on several EC2 instances and have them communicate with each other.

Once you have your code working, please follow these instructions for submitting your assignment:

- Export your Eclipse project to the file system.
- Create a zip archive file, named after you, containing a directory with your name. E.g. if your name is Humphrey Bogart, then name the directory `Humphrey_Bogart`.
- In that directory you should provide the zip archive that contains your complete Intellij project, and the server jar file, `dht.jar`.
- **Provide short videos demonstrating the successful testing of your instance**, with these scenarios as a minimum: adding listeners for remote bindings, adding bindings at a third site, adding a fourth node that causes a listened-to binding to be moved.
- Also include in the directory a filled-in rubric for your submission (see the submission Web page for the rubric).

It is very important for your grade that you do adequate testing. You should at a minimum demonstrate testing of these scenarios:

1. Joining at least three nodes together into a DHT network.
2. Adding bindings at both local and remote nodes. You can do all your testing using EC2, but use different instances behind their own firewalls. Use the `bindings` and `listeners` commands of the CLI to show the functioning of your network. You should demonstrate adding listeners for remote bindings, adding bindings at a third site, adding a node that causes a listened-to binding to be moved, and show that new bindings are still being reported at listeners after bindings have moved.
3. Record one or more short videos demonstrating your tool working. Upload these videos with your submission.
4. Display the state of your nodes at the beginning and end of the demonstration, including bindings and listeners.
5. Make sure that your name appears at the beginning of the video. For example, display the contents of a file that provides your name.

Finally note any other changes you made to the materials provided, with an explanation.

Remember the format of the submission: A zip archive file, named after you, with a directory named after you. In this directory, provide these files: a zip archive of your complete, a server jar file, a report (rubric) for your submission, and videos demonstrating your app working. Failure to follow these submission instructions will adversely affect your grade, perhaps to a large extent.