CSCI 4770 HW 3: Distributed Authorization Service

Assigned: Tuesday, Feb 26
Due: Tuesday, Mar 26, 11:00AM (LATE SUBMISSIONS NOT ACCEPTED)

1 Motivation

In the last two home work projects, we implemented a simple beacon service. First we located other beacons that are currently online and accessible. Next we implemented a beacon service that provided a search service to locate keys from remote beacons (that are not directly known to the current beacon).

Throughout the first two home work projects, beacons provided a mechanism for identifying themselves using a *passwd*. When a beacon logs into the system, it provided its *passwd*. On successful validation of this authentication key, a beacon is provided with an authorization token. This token was used for all further transactions. When a token was transferred across beacons, the tokens were assumed to be valid.

In this project, we will provide a simple authorization service. You will validate the authorization token issued for a user *passwd*. Note that each authorization token is unique across the system. In order to prevent a single token from being reused, each token should be valid for a fixed time interval. For ease of testing, assume that the tokens are valid for 1 minute. You may use a key server for validating an authorization token.

Your beacons will provide a new service that will utilize one or more key servers to validate authorization tokens. Such validation would be necessary if a beacon wants to provide service only to certain clients. Such validation would also allow the beacons to collect a fee for utilizing its services.

2 Description

The goal of this project is to provide a simple authorization service. For this project, the beacons will be extended to provide the following service:

- **authenticate(token)** The beacon authenticates the token and returns a new authorization token. Each new token is valid for a fixed time interval (set it to 60 seconds for ease of testing). Note than authentication of an invalid token will fail with an error. This operation can be used to revalidate tokens. A typical usage scenario is described below (user interactions are preceded by a $>$):

```
> open passwd
TOKEN: token123
> authenticate token123
TOKEN: token125
> searchget token125 PinkFloyd 2
PinkFloyd=Wall
> revoke token125
okay
> validate token125
TOKEN INVALID
> authenticate token123
TOKEN: token126
....
```

- **validate(token)** This service allows the user to validate if a token is valid. The current beacon will validate a token regardless of the beacon that created this authorization token. The beacon will print the passwd of the user to whom this token was issued, time left before token expiration or an error message if the token is invalid. The token may be invalid either because it was an illegal token or if the token had expired.

- **revoke(token)** The beacons can use this service to invalidate a token even before its time expires. For example, when a beacon has received the *key* that it was looking for, it could revoke the token.

Note that any of these operations can be performed on any beacon (and not necessarily on the beacon that created the token). You should also extend your beacon services (provided from Homework 1 and 2) to always validate a token (used in **GET**, **PUT**, **CLOSE**, **SEARCHGET** services) before using the token. Hence, services that used to work before might fail now because of an expired or an invalid token. **Note that any invalid token should always cause a particular operation to fail (e.g. GET, PUT, CLOSE, SEARCHGET, AUTHENTICATE, REVOKE, VALIDATE)**.

As usual, you are free to choose the exact technique to provide the service described above. You can use a single central server for key service or a distributed peer-to-peer approach for providing a robust service.

3 Submission

Please submit your project, along with a succinct report called **REPORT.txt** (plain text is fine) describing your approach, the merits of your approach and compilation instructions. You will turn in your complete project as a single tar file. On gemini, please use `/home/profs/surendar/bin/turnin UBICOMP HWP3 <your tar file>` to submit your assignment. You can submit your assignment multiple times. I will only use the latest submission. To see the files that you had submitted, try `turnin hw4`. **Remember, I will randomly choose students who will be asked to explain their approach in person**. Issues that you might consider while developing and evaluating your system are:

1. **robustness:** How reliable is your system against failures? Does your beacon recognize forwarding loops? (wherein the requests are forwarded around in a loop without making progress towards the destination)

2. **scalability:** If your beacon suddenly becomes popular because of the files that your provide, how much load can you tolerate before your system crashes? Does your service degrade gracefully? Are you immune to denial-of-service attacks? (wherein, beacons repeatedly open connections to you to prevent you from servicing other, legitimate users)