CS855 ATM Server Documentation

Release 1.0

Derek Weitzel

April 21, 2013

CONTENTS

This is a programming assignment for CS855 Distributed Operating Systems class.

Contents:

CHAPTER

ONE

DESIGN DOCUMENT

This document will discuss the ATM RPC design.

1.1 How it Works

The ATM is designed using the python XMLRPC library.

In my design, the `ATMClient` directly connects to the server. In this case, the `ATMClient` must know the Server's address. Full documentation of the Client's operation is documented on the *Client Documentation*.

The Server exports an instance of the `ATMServer` object. This is done with the code:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

server = SimpleXMLRPCServer(('', 9000), logRequests=True, allow_none=True)
server.register_introspection_functions()
server.register_multicall_functions()
server.register_instance(ATMServer())
try:
    print 'Use Control-C to exit'
    server.serve_forever()
except KeyboardInterrupt:
    print 'Exiting'
```

Each public function (funciton not starting with an underscore) in the `ATMServer` is then available to be called. Additionally, state is kept between calls since it's all one instance.

The `ATMServer` starts with only 2 accounts. Since there is only 2 accounts, responses are only give for 2 accounts. All requests for other accounts will cause an error.

The RPC functions all return a special data structure, a *Status Dictionary*, which will return the status of the request, as well as a failure reason or a request response. This is transferred with a dictionary, which is then serialized by the underlying XML RPC library.

In addition to a fail reason or request response, the *Status Dictionary* can also respond with a WAIT command. The WAIT command will tell the `ATMClient` to wait while the account is locked.

1.2 Transactions

When a client connects to the `ATMServer`, the first attempt to access an account will cause it to attempt to lock the account. The `ATMClient` will do nothing on `begin_transaction`, since it's not accessing an account.

When the client sends an `commit` or `abort` to the server, it will go through each account the customer has a lock on and either commit or abort the transaction on the account, and remove the lock on the acocunt.

When a customer requests a lock on a account, if the account is already locked, the customer will be placed on a FIFO queue and the client is told to `WAIT`. Only the next customer on the queue will be allowed to grab the lock on the account the next time it is requested.

Further, the account locking is thread safe, using mutexes protecting the locking and unlocking code.

1.3 Design Trade-offs

he XML-RPC library provided by Python greatly simplifies the RPC aspect of the communication. This is because the Python XMLRPC:

- Does not require interface definition
- Does not require a registrer
- Performs simple XML HTTP requests in order to perform RPC.

Because of these advantages over more complex RPC systems, such as Java's RMI, it is much simplier to create a RPC server in Python.

Since the requests are all done over HTTP, it is very easy and well known how to secure the communication (HTTPS) and how to load balance requests (Linux Virtual Server).

The `ATMClient` is fully stateless. The `ATMServer` holds all state on which customer holds locks on each account, and the transaction changes to the accounts.

1.4 Improvements and Extensions

The design of the `ATMServer` is very simple. It could be improved by:

- Secure Account Storage - Since we are running on Amazon EC2, this could easily be done with on of many database systems available on EC2 such as Dynamo and RDS.
- Load Balanced Servers - Many more clients can be handled with load balanced HTTP requests.
- Authenticated Requests - It is well known how to handle authorize HTTP requests, digest or SSL Certs.

CHAPTER

TWO

SERVER DOCUMENTATION

2.1 Status Dictionary

All public functions return a status `dict` with the following elements:

status (str) Either `OK`, `ERROR`, or `WAIT` indicating success, failure, or telling the client to wait and try again, respectibly.

reason (str) If status is `ERROR`, then reason is the error string describing the issue.

result (str) If the function requires sending back information, it will be stored in the result.

2.2 Operation

This is the main ATMServer that should be executed.

It can be executed with the command:

```
$ python ATMServer.py
```

By default, it will run on port 9000 of the local server, and will listen to traffic from all sources (insecure!)

2.3 ATMServer Technical Document

class `ATMServer.`**`ATMServer`**

The Main ATMServer that should be used to export.

`abort_transaction`(*customer*)

Abort a transaction for the customer

Parameters **customer** (*int*) – Customer number

Return type status dict, see: *Status Dictionary*

`commit_transaction`(*customer*)

Commit a transaction for the customer

Parameters **customer** (*int*) – Customer number

Return type status dict, see: *Status Dictionary*

`deposit`(*customer*, *account*, *amount*)

Deposit money into the account.

Parameters

- **customer** (*int*) – Customer number
- **account** (*int*) – Account number to deposit money into
- **amount** (*float*) – value to be added to the account

Return type status dict, see: *Status Dictionary*

This function will fail if the account does not exist.

`inquiry` (*customer*, *account*)

Inquire the account amount

Parameters

- **customer** (*int*) – Customer number
- **account** (*int*) – Account number to deposit money into

Return type status dict, see: *Status Dictionary*

`withdraw` (*customer*, *account*, *amount*)

Withdraw money from the account

Parameters

- **customer** (*int*) – Customer number
- **account** (*int*) – Account number to deposit money into
- **amount** (*float*) – value to be added to the account

Return type status dict, see: *Status Dictionary*

This function will fail if the account does not have sufficient funds to make the withdraw.

class `ATMServer.Account` (*id*, *amount*)

A lockable account in the ATM that will follow transactions.

`abort` ()

Abort the pending transaction

`commit` ()

Commit the pending transaction

`deposit` (*amount*)

Deposit amount into the the account.

Parameters **amount** (*float*) – value to be added to the account

`inquiry` ()

Inquiry the account

Return type float the amount of money in the account

`lock` (*customer*)

Lock the account. If successful, will return True, otherwise False.

If the lock has been set by another customer, the current customer will be put on a queue and when the customer asks again for the lock, it will receive the lock if it is next on the FIFO queue.

Parameters **customer** (*int*) – Customer id number

Return type boolean if locking is successful

`unlock`(*customer*)

Unlock the account.

Parameters **customer** (*int*) – Customer id number

`withdraw`(*amount*)

Withdraw money into the account

Parameters **amount** (*float*) – value to be added to the account

CHAPTER

THREE

CLIENT DOCUMENTATION

The ATM Client can be used for example by issueing the command:

```
$ python ATMClient.py <server> <port> <operation> <customer> [<account> [<amount>]]
```

server The server that the ATM client should connect to.

port Port that the client should use to connect to the ATM Server

operation One of:

- inquiry - Check the status of an account
- deposit - Deposit money into the account
- withdraw - Withdraw money from an account
- begin_transaction - Start a transaction for a customer
- end_transaction - Commit a transaction for a customer
- abort_transaction - Abort a transaction for a customer

customer This is the customer number. Must be an integer.

account This is the numeric account number. Must be an integer

amount Amount of money to either deposit or withdraw

CHAPTER

FOUR

EC2 EXECUTION

EC2 Execution is done by submitting jobs to HTCondor that start EC2 Spot Instances.

4.1 Preparing Submission

Submission is done with HTCondor. An EC2 Submission file was created:

```
universe = grid
grid_resource = ec2 https://ec2.amazonaws.com/
executable = /bin/true

# Amazon Linux AMI
ec2_ami_id = ami-04cf5c6d

ec2_access_key_id = access_key
ec2_secret_access_key = secret_key
ec2_keypair_file = keyfile.$(CLUSTER)
ec2_security_groups = bosco
ec2_instance_type = m1.small
ec2_spot_price = 0.04
ec2_user_data_file = ami-run.sh
ec2_tag_concat = true
periodic_remove = JobStatus == 2 && (time() - EnteredCurrentStatus) > 60*120
log = ec2.log
queue
```

As you can see, `ec2_user_data_file` is set to ami-run.sh. This ami-run.sh contains the script that will be executed at the first start of the instance:

```
#!/bin/sh

rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
yum -y install yum-priorities git

# Checkout the project
git clone git://github.com/djw8605/CS855PA1.git

# Start the server
cd CS855PA1
git checkout PA3
python src/ATMServer.py
```

Each instance will run this script since the Amazon Linux AMI contains CloudInit.

Addionally, the submit file has a `periodic_remove` statement that will kill the instance after 2 hours of execution.

4.2 Testing Execution

Testing was done by SSHing into the EC2 instance and running the Clients against another EC2 instance:

```
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 inquiry 1 100
Current value of account 100 is $1000.00
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 deposit 1 100 200
Successful deposit of $200.00 to account 100
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 inquiry 1 100
Current value of account 100 is $1200.00
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 abort_transaction 1
Customer 1 aborts their transaction
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 begin_transaction 1
Customer 1 starts their transaction
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 inquiry 1 100
Current value of account 100 is $1000.00
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 deposit 1 100 200
Successful deposit of $200.00 to account 100
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 withdraw 1 100 50
Successful withdraw of $50.00 to account 100
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 inquiry 1 100
Current value of account 100 is $1150.00
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 end_transaction 1
Customer 1 commits their transaction
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 begin_transaction 1
Customer 1 starts their transaction
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 inquiry 1 100
Current value of account 100 is $1150.00
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 deposit 1 100 200
Successful deposit of $200.00 to account 100
```

And on C2:

```
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 begin_transaction 2
Customer 2 starts their transaction
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 inquiry 2 200
Current value of account 200 is $1000.00
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 inquiry 2 100
<blocking...>
Current value of account 100 is $1350.00
$ python ATMClient.py ec2-54-224-134-194.compute-1.amazonaws.com 9000 abort_transaction 2
Customer 2 aborts their transaction
```

This testing shows that the blocking and locking mechanism works.

PYTHON MODULE INDEX