Ben Jones

Spring 2012

DDOS Failover API Interface

Special note: I have not listed return values for most of these methods because Ruby just returns the value of the last statement from any method

Class DDOS\_API methods

1. accessor methods: each function returns or sets the value of the variable with the same name
   1. get\_failover\_connection
   2. get\_failover\_timeout
   3. get\_service\_object
   4. set\_service\_object=
2. threading function-> functions just used to implement multi-threading
   1. accept\_thread(TCPServer server)
      1. this method takes an open TCPServer connection and accepts incoming connections to the socket. Opened connections are added to the connection list, time-stamped for their last received data, and lets the input thread handle the data
   2. backup\_thread()
      1. will force a backup of the service and send the backup to all @failover\_connects if that connection has not been backed up in @force\_backup seconds. This is designed so that if the service forgets to backup, all data will be sent over but it also won't overwrite backups from checkpoints if they were sent in the last @force\_backup seconds
   3. input\_thread(null)
      1. this method uses the instance variable @receive\_connections to determine if there is input pending on any of the connections. If no data has been received for @failover\_timeout seconds from that connection, and the current server is monitoring that connection, then it will use fail\_server to failover that server and notify all of the dependent servers of the switch.
3. initialize(int failover\_timeout, string local\_ip, int local\_port, service\_object)
   1. constructor for DDOS\_API class that sets up the timeout in seconds, the host name/ ip address for local server, the local port number, and a takes a reference to an instantiated object that is using the API. service\_object should contain methods to\_yaml to convert the object to yaml for backup and get\_data to deal with incoming data.
4. config\_api(string failover\_filename)
   1. method to begin accepting threads and read in failover configuration from a text file. The default configuration file is failover.txt and will be used if no filename is provided. The syntax specifications for the configuration file are given under the failover\_filename variable. This method will initiate an input thread and a thread to accept new connections on the one given local ip address and port.
5. Public Methods
   1. accept\_fail(string address, port)
   2. check\_status(nil)
      1. will check the status of all connections on the status\_unsure list and take appropriate action. If the connection has not received any input in @failover\_timeout seconds, or it has been marked dead, then this method will perform additional logic to determine how to failover the connection and if the local server should act as a backup/ hot failover server.
   3. failover\_server(int source\_serv, int dest\_serv, int rec\_serv)
      1. will instruct the server at the connection pointed to by rec\_serv to use dest\_serv instead of source\_serv in its connections. If the local server is source\_serv or dest\_serv, the server will notify its dependent servers of the change.
   4. kill(nil)
      1. will set @@kill to true and suspend all network communication for the local server
   5. unkill(nil)
      1. will set @@kill to false and re-enable network communication for the local server
   6. proc\_alive(int index)
      1. will process incoming alive messages, update time stamps, and remove the server from the @status\_unsure array
   7. proc\_fail(int index, string request)
      1. will process incoming failover request and will either consider itself DDOSed and change @alive or will adjust its connections to use the destination server over the origin server
   8. receive\_data(int index)
      1. process data from the index connection of the @rec\_connects array and determine what to do with the data. If the data should not be processed by a kill, backup, or failover method, then the data is handed to the get\_data method of the service object
   9. receive\_backup(string backup\_data)
      1. pull the time stamp from the backup and if it is newer than the last backup from this connection, then replace the old backup with this data. Returns 1 if the received backup was used and 2 if the existing backup was kept.
   10. self\_fail(nil)
       1. the local server has been DDOSed and the server will now attempt to inform all its dependent servers that they should switch to the backup. It does this by calling failover\_server on all its dependent servers
   11. send\_alive(int index)
       1. this method will send a message to another server asking if it is still alive. The code in the input thread will use this method to query another servers state, then add it to a list of connections to check in @failover\_timeout seconds.
   12. send\_backup(int id, string backup\_data)
       1. performs the actual sending of backup data to the destination connection, referred to by its id
   13. send\_backup(int id)
       1. send a backup of the service object to the id connection in @send\_connects. This method will just call the to\_yaml method of the service object and then send the marshalled data to the connection along with a time stamp for the backup.
   14. send\_backup(null)
       1. send a backup of the service object to all of its backup connections in @send\_backups. Just calls the send\_backup(int, string) method for each backup connection. Designed to be used by the service when it wants to back all its connections up or it is unaware of what connections are hot failovers.
   15. send\_data(int connect\_id, array proc\_data)
       1. send proc\_data to the previously opened connection in the @connects array referred to by connect\_id. If the server has been killed, then this method will return nil
   16. send\_data(string address, int port\_num, array proc\_data)
       1. send whatever data the local server gives via proc\_data to a previously unopened connection described by address and port\_num. Just opens connection and then uses rest of send\_data functionality described above
   17. send\_fail(int fail\_from, int fail\_to, int rec\_serv)
       1. the method will send a message containing failover information to another server. Fail\_from contains the id of the server that has failed, fail\_to contains the id of the backup server, and rec\_serv contains the id of the server that the message will be sent to. If fail\_from or fail\_to are set to -1, then the local address and local port information are used to send the message.
6. Private methods
   1. create\_connect(string address, int port)
      1. will create a connection to the server on the given address and port. This method will return an array of the index and the id in the form [index, id]
   2. delete\_elem(int index, reference \* list)
      1. will delete all 5 elements for that index on the list
   3. find\_all\_by\_id(int id, reference \* list)
      1. will return an array of all the connections with the given id in the given list.
   4. find\_all\_by\_ip(string address, int port, reference \* list)
      1. will return an array containing all indices to connections with matching ip and port information in the list
   5. find\_id\_by\_ip(int id)
      1. will return the ip and port associated with this id in the form [ip, port]. If the id is not in the table, then return nil
   6. find\_index\_by\_id(int id, reference list)
      1. will find the index of the connection with the given id. If the connection is not found, the method will return nil
   7. find\_index\_by\_ip(string address, int port, reference \* list)
      1. will find the index of the connection to the socket given by address and port in the connection list given by list. If the connection is not found, the method will return nil
   8. find\_ip\_by\_id(int id, reference \* list)
      1. will return ip and port of the connection with the given id in the list
   9. get\_id\_by\_ip(string ip, int port)
      1. will return the id of the connection with the given ip and port. If the id is not found, this function will create and return a new id
   10. mark\_dead(int id)
       1. will mark all connection with an id of id as dead and close the connections

Class DDOS\_API Variables

1. Connection arrays: each list is an array of sub-arrays with the 0th element for the connections, the 1st element for the addresses, the 2nd the ports, the 3rd element for time stamps, the 4th element stores an ID, and the 5th element stores a boolean true or false indicating whether the server is alive. This set up is fairly awkward, but it allows me to easily determine what connections have pending input.
   1. Note: the connections that use mutual exclusion each have an order listed starting with 1. If you are using multiple of these resources at once, this is the order in which you must request the resources to prevent priority inversion
   2. @connects
      1. array of who the service is receiving data from
      2. includes the backup up as the 0th element, a time stamp from when the backup was made on the remote server as the 3rd element, the connection id as the 4th element, and true or false as the 5th element to show whether the connection is still alive or not
      3. has mutual exclusion order 6
   3. @dead\_connects
      1. array of connections that have been marked as dead. This is not used in this implementation of the API, but I plan to add support to add failed servers back into the failover list in future versions of the API
      2. has mutual exclusion order 1
   4. @dependents
      1. array of who is dependent upon the local service
   5. @fail\_rec
      1. array of who the local service should monitor and be a hot failover for
   6. @fail\_to
      1. array of who I could failover to with their connection info, ids, and time stamps
      2. has mutual exclusion order 4
   7. @rec\_backups
      1. array of backups received with their time stamps
      2. has mutual exclusion order 3
   8. @status\_unsure
      1. array of connections that may be dead with a time stamp for when they should be considered dead
      2. has mutual exclusion order 2
2. Local Service Status and Identity
   1. @alive
      1. stores whether the API is DDOSed or not
   2. @local\_ip, @local\_port
      1. stores the address and port number that I am receiving connections on
   3. @@kill
      1. stores whether the DDOS API is shutdown or not
3. Configurable Parameters
   1. failover\_filename
      1. used in the config\_api function, this configuration stores the information for who we should failover to and who we are monitoring. All data for a server should be on the same line, and the line should be formatted with the local server's info, then the connections to monitor(fail\_rec), and then the connections who will be a failover for this server(fail\_to). This data should be entered in the format “local\_ip local\_port fail\_rec monitor\_ip1 monitor\_port1 monitor\_ip2 monitor\_port2 fail\_to back\_ip1 back\_port1 back\_ip2 back\_ip2” or in BNF
         1. failover configuration file format: <config file> ::= <local info> “fail\_rec” <monitor info> “fail\_to” <backup info>
   2. @failover\_timeout
      1. how long to wait in seconds before concluding that the local server has been DDOSed
   3. @force\_backup
      1. how long to wait in seconds before forcing the local server to override checkpointing and make a backup
   4. @service\_object
      1. alias to the service using this instance of the API
      2. this object must contain several methods, load\_fail to load a backup into the service to use when taking over in case of DDOS, get\_data to handle incoming data, and to\_yaml to backup data
         1. get\_data(string data, int connect\_index)
         2. to\_yaml(null)
         3. init\_fail(reference \* data)
4. Assorted Variables
   1. @backups
      1. array of backups received along with a time stamp
   2. @accept\_threadT, @backup\_threadT, @input\_threadT
      1. store references to the threads should they be needed