BULLETIN (that) NEVER GIVES SECURITY
BNGS

PROTOCOL SPECIFICATION

November 2021

prepared for
Dr. Edward Zieglar
CMSC 481 - Computer Networks
University of Maryland, Baltimore County
1000 Hilltop Circle
Baltimore, Maryland 21250

by
Drew Barlow
University of Maryland, Baltimore County
1000 Hilltop Circle
Baltimore, Maryland 21250

TABLE OF CONTENTS

INTRODUCTION

1.1 | Motivation

The motivation for the development of the BNGS protocol is obvious; it is an assignment. We wish to develop a protocol such that any number of clients may connect to a server and post messages to a bulletin board. The acronym is inspired by Bingus the cat.

1.2 | Description

The Bulletin (that) Never Gives Security protocol, henceforth referred to as the BNGS protocol, provides simple message-sharing functionality over bulletin boards. It allows for users to post public messages, able to be seen by all, and private messages, able to be seen by groups of users. Users may also create groups, remove their own messages, and delete groups that they have created. Each message has a subject line, as well as a content body. This protocol is built upon the Transmission Control Protocol.

1.3 | About This Document

This document's formatting is very heavily formatted after RFC 793.

FUNCTIONALITY & DOCUMENTATION

2.1 | Description

The BNGS protocol and its client/server derivations are implemented in Python 3. It uses the socketserver built-in library to handle threaded connections, and the typing built-in library for explicit typing. It does not make use of any external libraries. No security is provided. BNGS is stateless in that it does store user information between sessions. If a user creates a group or message, they are unable to remove them after disconnecting and reconnecting.

2.2 | Server

The server is implemented in the BNGS.py and request_handler.py files. It handles all connections, queries, bulletins, users, groups, and messages. Connectivity is implemented over TCP.

2.3 | Client

The client file is very sparse. It simply connects to the server, sends input, receives output, and displays information accordingly.

2.4 | The BNGS Query Language

The BNGS protocol makes use of a very simple and barebones querying language. Three query prefixes exist, with two or four commands derived from each prefix. However, eleven different commands exist. Queries are parsed and executed on the server-side. The parser is implemented inside of server/src/lang/parser.py, and uses the recursive-descent parsing technique.

2.4.1 | Query Prefixes

ADD

Queries prefixed by “ADD” will carry out instantiation of either a group or message.

REMOVE

Queries prefixed by “REMOVE” will erase already-existing groups from the bulletin or messages from a group.

GET

Queries prefixed by “GET” will retrieve message data from groups.

2.4.2 | Queries

ADD message

Prompts a user for a subject line and body. Instantiates a message with gathered information, including author information, and adds the message to a group of the author's choosing.

ADD group

Instantiates a new group with the author's information. Returns to the author the new ID of the group.

REMOVE message

Prompts a user for the subject line of the message they wish to delete. Also prompts for the group in which the message resides. Deletes the message if it exists.

REMOVE group

Prompts a user for the group ID they wish to Delete. Deletes the group if it exists and the user is the owner.

GET subjects

Prompts the user for a group ID to request subject lines from. Returns a nicely formatted list of all subject lines from the requested group. Marks and displays unseen messages with a noticeably different formatting than previously seen messages.

GET message

Prompts the user for the subject line and group of a message they wish to view. Returns the formatted message to the user.

GET messages

Prompts the user for a group ID to request messages from. Returns a nicely formatted list of all messages, subject lines, and their authors from the requested group. Marks and displays unseen messages with a noticeably different formatting than previously seen messages.

GET NUM messages

Prompts the user for a group ID to request total messages from. Returns the number of messages in a formatted string to the user.

GET NUM new

Prompts the user for a group ID to request the number of new messages from. Returns the number of new messages in a formatted string to the user.

HELP

Displays all commands with short descriptions to the user.

BYE

Closes the connection to the server on the client's side.

2.5 | Classes

2.5.0 | PREFACE

This documentation is structured in strongly-typed Pythonic syntax. As a result:

- -Attribute/Function names with a leading underscore are considered PRIVATE.
- -Listed Attributes are local variables or constants, scoped to the class they are listed under. Their syntax is as follows:
  variable_name: *data_type* = default_value
- -Listed Functions are local to the class they are listed under. Their syntax is as follows:
  For functions with no parameters:
  function_name(*None***)** -> *return_type*
  For functions with parameters:
  function_name(param_name: *param_type***)** -> *return_type*
  For functions with default parameters:
  function_name(param_name: *param_type* = param_value**)** -> *return_type*

2.5.1 | BNGS

The implementation of the server functionality lies in the BNGS class.

Attributes:

_server: *ThreadingTCPServer* = None
Stores the ThreadingTCPServer object. Initialized inside of start_server().

_HOST: *str* = host
Customizable host IP constant.

_PORT: *int* = port
Customizable host port.

Functions:

__init__(host: *str* = "127.0.0.1", port: *int* = 13037**)** -> *None*
Raises: N/A
Constructor with default params.

start_server(*None***)** -> *None*
Raises: N/A
Initializes and starts the ThreadingTCPServer to the requested ip and port, with a custom connection handler defined in the Handler class.

2.5.2 | Handler

The Handler class inherits from socketserver.StreamRequestHandler. It is instantiated upon every connection to the ThreadingTCPServer. It handles every connection and its queries.

Attributes:

static **_data**: *dict* = {
"bulletin": Bulletin()
"users": [],
"last user id": -1
}

Declared and initialized outside of the constructor. A dictionary is chosen to store the necessary working server information due to its static properties. The same data is shared among all instances of the Handler class because of this.

Functions:

handle(*None***)** -> *None*

Raises: N/A

Overrides socketserver.StreamRequestHandler.handle(). This is the main function of this class. It handles User registration, queries, and error handling in regards to closed connections.

_parse(query: *str***)** -> *bool*

Raises: N/A

Makes use of a query parser specified in section 2.4 to parse user input. Returns True if the user inputs a valid query, False otherwise.

_det_query(query: *str*, user: *User***)** -> *None*

Raises: N/A

If the parsed query is valid, then this function decides which query function to send control to.

_query_help(*None***)** -> *None*

Raises: N/A

Sends the HELP message specified in section 2.4.2 to the user.

_query_add(target: *str*, user: *User***)** -> *None*

Raises: N/A

Carries out ADD functionality specified in section 2.4.1. Informs users of success/failure in adding a message to the bulletin or creating a new group. Handles necessary exceptions associated with this functionality.

_query_remove(target: *str*, user: *User***)** -> *None*

Raises: N/A

Carries out REMOVE functionality specified in section 2.4.1. Informs users of success/failure in deleting a group or message. Handles necessary exceptions associated with this functionality.

_query_get(tokens: *List[str]*, user: *User***)** -> *None*

Raises: N/A

Carries out GET functionality specified in section 2.4.1. Displays subject lines, messages, or requested numerical values as requested by the user. Handles necessary exceptions associated with this functionality.

_new_message_details(*None***)** -> *Tuple[str, str, int]*

Raises: N/A

Called by _query_add() when the request to create a new message is made. Requests user input for a new message's subject line, content, and group ID. Returns this information in a tuple as such: (subject, content, group_id).

_old_message_details(*None***)** -> *Tuple[str, int]*

Raises: N/A

Called by _query_remove() when the request to remove an old message is made. Requests user input for the existing message's subject line and group ID. Returns this information in a tuple as such: (subject, group_id).

_group_details(*None***)** -> *int*

Raises: N/A

Called by most _query_...() and all _..._message_details() functions, respectively. Requests user input for a group ID. This is received as a string and promptly converted to an integer. If the input is an invalid integer, then the group ID is defaulted to None; the ID of the public group.

2.5.3 | User

The User class defines the structure by which each client is identified on the server.

Attributes:

_id: *int* = id
Upon successful connection, a user is assigned a unique, immutable identifier to distinguish them from other users.

_username: *str* = name
A user may choose a display name to identify themselves across bulletin boards. Multiple users may share the same name; BNGS does not care.

_seen_messages: *Dict[int, List[int]]* = {}
Stores the seen message ids of each group. The dictionary consists of key-value pairs, where the key is the group id, and the value is a list of message ids seen by a user from that group.

Functions:

__init__(id: *int*, name: str**)** -> *None*
Raises: N/A
Constructor.

get_seen_messages(g_id: *int***)** -> *List[int]*
Raises: N/A
Returns the list of seen message ids from a group with g_id.

seen_message(g_id: *int*, m_id: *int***)** -> *None*
Raises: N/A
Adds a message id, m_id, to the list of seen messages for a group with g_id.

get_info(*None***)** -> *Tuple[str, int]*
Raises: N/A
Simple accessor function. Returns a Tuple consisting of this User's _username and _id, in that order.

2.5.4 | Group

The Group class is able to be instantiated by Users. Each Group may contain Messages. Groups can be deleted by the User that created them.

Attributes:

_id: *int* = id
Each Group has a unique identifier. This identifier is used as a "password" of sorts when trying to access a private bulletin.

_messages: *Dict[str, Message]* = {}
Holds all of this Group's Messages as key-value pairs. No two Messages may have the same subject line.

_last_message_id: *int* = -1
Used to keep track of Message identifiers. Backtracking to save on space is not considered at this time. That is, if a Message with _id 5 is removed, 5 will not be reused as an identifier in this Group.

_owner: *User* = owner
The User that created this Group is stored for verifying ownership when a deletion of this Group occurs.

Functions:

__init__(id: *int*, owner: *User* = None**)** -> *None*
Raises: N/A
Constructor.

get_info(*None***)** -> *Tuple[int, User, int]*
Raises: N/A
Returns _id, _owner, and the number of _messages.

post_message(subject: *str*, content: *str*, author: *User***)** -> *None*
Raises: Exceptions.DuplicateSubjectLine
Instantiates a new Message and adds it to _messages with the key being the subject line. New unique identifier (_last_message_id + 1).

del_message(subject: *str*, requester: *User***)** -> *Message*
Raises: Exceptions.UserIsNotOwner
Verifies that the User attempting to delete some Message is that Message's author. Raises an exception if unsuccessful, returns the deleted Message if successful.

get_fmt_message(subject: *str*, user: *User***)** -> *str*

Raises: Exceptions.MessageDoesNotExist

Returns a formatted version of the message with requested subject line.

get_fmt_subject_lines(user: *User***)** -> *str*

Raises: N/A

Returns the subject line of every Message posted to this group. Returns "***empty***" if there are none. If a user has not seen any messages, it will output "**NEW**" before each unseen subject line.

get_fmt_messages(user: *User***)** -> *str*

Raises: N/A

Returns the subject line, content, and author of every Message posted to this group. Returns "***empty***" if there are none. If a user has not seen any messages, it will output "**NEW**" before each unseen message, then add this message to the user's seen messages for this group.

get_num_new_messages(user: *User***)** -> *str*

Raises: N/A

Returns the number of unseen messages by the user in this group in a formatted string.

2.5.5 | Bulletin

The Bulletin class is a simple container for everything message or group related. In simple terms, Bulletin manages everything other than connections.

Attributes:

_last_group_id: *int* **=** -1
Each group has a unique ID. The Bulletin, which is responsible for creating and managing groups, uses an incremented integer to assign group IDs.

_groups: *List[Group]* = [Group(None)]
Used to store all Groups on the Bulletin.

Functions:

__init__(*None***)** -> *None*
Raises: N/A
Constructor.

get_group(id: *int* = None**)** -> *Group*
Raises: Exceptions.GroupDoesNotExist
Searches through _groups with requested id. Returns the group if it exists, throws GroupDoesNotExist otherwise. The default public group has

create_group(user: *User* = None**)** -> *Group*
Raises: N/A
Creates a new Group with the user as the author; appends it to _groups. Returns the newly created group.

delete_group(user: *User*, id: *int***)** -> *Group*
Raises: Union[
Exceptions.GroupDoesNotExist,
Exceptions.UserIsNotOwner
]
Removes a group from the board and returns it.

2.5.6 | Message

The Message class contains several fields of data. Any User may instantiate a Message through a Group. Any User may also remove a Message through a Group.

Attributes:

_id: *int* = -1
Each Message has a unique identifier local to the Group it is posted to.

_subject: *str* = subj
The subject line of a Message provides a general summary of what the Message's content contains.

_content: *str* = content
A user can input whatever information they would like to in the content of a message.

_author: *User* = author
The Author is stored in every Message for easy checking when deletion occurs.

Functions:

__init__(id: *int*, subj: str, content: str, author: *User***)** -> *None*
Raises: N/A
Constructor.

get_subject(*None***)** -> *str*
Raises: N/A
Returns _subject.

get_content(*None***)** -> *str*
Raises: N/A
Returns _content.

get_author(*None***)** -> *User*
Raises: N/A
Returns _author.

2.5.7 | Exceptions

class DuplicateSubjectLine(*Exception*)
Raised when two messages with the same subject line exist in the same Group.

class GroupDoesNotExist(*Exception*)
Raised when a specified Group cannot be found.

class MessageDoesNotExist(*Exception*)
Raised when a message with a specified subject line cannot be found.

class UserIsNotOwner(*Exception*)
Raised when comparison of two Users fails, usually when attempting to delete a Group or Message.

2.6 | Error Conditions

Error conditions are mainly handled by the custom exceptions defined in section 2.5.7. When an exception is raised, such as a user attempting to delete another user's message, the text contained inside of the exception will be sent directly to them as a detailed description of what they were not allowed to do. Otherwise, successful transactions will respond to the user with "OK" or their requested query response.