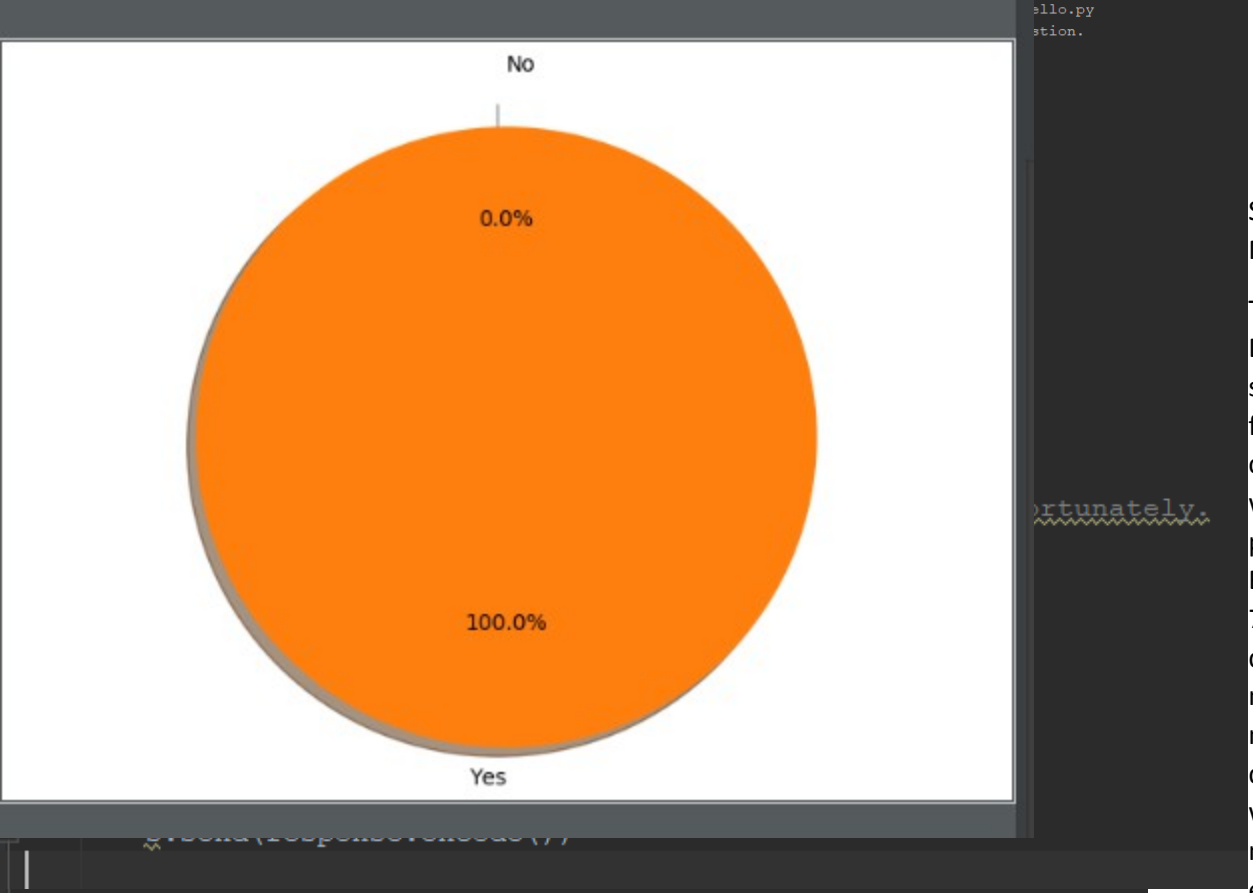


Status of This Memo:

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. This document may not be modified, and derivative works of it may not be created, except to publish it as an RFC and to translate it into languages other than English.

```
print("Attempting Connection.")
conn, addr = s.accept()
clientNum = str(addr[1])
print("Connected with " + addr[0] + " : " + str(addr[1]))
if not adminList:
    adminList.append(conn)
    start_new_thread(adminthread, (conn,))
else:
    clientList.append(conn)
    start_new_thread(clientthread, (conn, clientNum))

s.close()
```

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html

This Internet-Draft will expire on Fail 1, 0000.

1. Table of Contents
2. Introduction

3. Basic Information
4. Structure
    a. Server
    b. Administrator
    c. Client
    d. Usage:
        d.i. Clear
        d.ii. Poll
        d.iii. Response
        d.iv. Show
5. Semantics & Restrictions
6. Connections & Closing

2. Introduction:

The class polling tool is meant to serve as the groundwork for an in-class polling application. The inspiration for the tool was seeing professors request feedback from students, only for students to be too shy to respond and ask questions. This program is meant to serve as the messaging layer that a graphical application layer would sit on. Thus, for demonstration purposes, the messaging options are kept purposefully short.

The overall architecture has three components:

A. The **server side architecture** continually runs, and houses the run-time instructions for the server, the administration-server side, and the client-server side. Thus, the server must be run first as that provides the instruction set and interface for the administrator and the client.

B. The **administrator** has a client side. The server grants the first connection administrator privileges, and all future clients are standard clients. When the client side connects with the server, the server creates a separate administrator thread. The administrator can ask questions, request votes, show votes, and clear for new questions.

C. The **standard client** (also known as the **student**) has a standard yes/no voting interface. The student simply votes yes and no in response to in-person questions from the teacher. Their vote will be stored in their own data structure on the server side to allow for quick vote tallying.

3. Basic Information:

All communication described in the protocol occurs over TCP/IP. Both the client side and the server side code is written in Python. The first connection from the client side to the server side is automatically

granted administrator privileges. This is both for simplicity, as well as assuming that the teacher controls when the server will be activated. All subsequent connections are provided student controls.

The connections occur over persistent HTTP, and happens on port 4444. Non-administrator clients are free to disconnect or connect as they please. It is important to note that in this demonstration version, client votes are stored using a non-persistent key – the port number integer value from the client address tuple. Because of this method of voter identification, clients have the ability to stuff the ballot box by connecting, voting, disconnecting, then reconnecting and receiving a new, discrete port number. In any production version of this application, a more robust user identification and authentication system would be required in order to prevent this.

4. Code Structure:

```
# Host information
host = '10.0.0.175'
port = 4444

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print("Socket created...")

try:
    s.bind((host,port))
except:
    print('Binding failed')
    sys.exit()

print("Socket bound...")

s.listen(100)

print("Socket is listening...")

while True:
    print("Attempting Connection.")
    conn, addr = s.accept()
    clientNum = str(addr[1])
    print("Connected with " + addr[0] + " : " + str(addr[1]))
    if not adminList:
        adminList.append(conn)
        start_new_thread(adminthread, (conn,))
    else:
        clientList.append(conn)
        start_new_thread(clientthread, (conn, clientNum))
s.close()
sys.exit()
```

Server (Above)- When the server is started, it creates a socket and listens on port 4444. If the adminList is empty (i.e., if there is no admin), the first connection is routed to the admin thread.

```
# Admin thread function for first thread created
def adminthread (conn):
    has_exited = False
    conn.send("Welcome, administrator!\nAll connected parties will be able to vote
    while not has_exited:
        has_exited = adminmenu(conn)
    server_shutdown = True
    conn.close()
```

The admin thread (previous image) checks the boolean status of a local variable in order to set up menu recursion. If the admin menu (next image) has set that, the adminthread will set the server shutdown to True, initiating a server shutdown, and then closing its own socket. If not, it will call the admin menu function until such a time as has_exited becomes true.

```
def adminmenu (conn):
    conn.send("\n\nTo ask a new question, type 'Clear'\nTo show the results,
    menu_input = conn.recv(1024)
    menu_input = (str(menu_input.decode().lower()))
    if menu_input == 'clear':
        clear(votes)
        conn.send("Clearing voting data.\n".encode())
        return False
    if menu_input == 'show':
        show(votes)
        return False
    if menu_input == 'quit':
        for client in clientList:
            client.shutdown(socket.SHUT_RDWR)
            client.close()
        for admin in adminList:
            admin.send("shutdown".encode())
            admin.shutdown(socket.SHUT_RDWR)
            admin.close()
        return True
    else:
        conn.send("Invalid input!".encode())
        return False
```

Admin- The admin thread loops in a continuous loop over the admin menu function, which allows the user a few key options. Namely, clear to clear all current votes and starts a new question; show, to show the current results using matplotlib, and quit to exit both the recursive menu function and the program.

The admin menu function is the redirection for the client side admin to provide input commands and actions. It listens for a response from the client, acknowledges the response, executes the request, and then closes out, waiting to be executed again.

```
# Client thread function called for each new thread after the first
def clientthread(conn, clientNum):
    while server_shutdown == False:
        data = str(conn.recv(32).decode().lower())
        if not data:
            break
        print(clientNum, data)
        if data == 'yes':
            votes[clientNum] = 'yes'
            print(votes)
        elif data == 'no':
            votes[clientNum] = 'no'
            print(votes)
        else:
            pass
    clientList.remove(conn)
    conn.shutdown(socket.SHUT_RDWR)
    conn.close()
```

The clientthread function is created and called when a new non-admin thread is created (student). On the server side, the client thread is created and the clientthread function is called, creating a loop for the client. If the client votes yes, then their vote entry is changed to yes. If they vote no, their vote is recorded as no. This information is stored as a key:value pair in a python dictionary, with the client port number functioning as the key and the vote as the value. Any entry other than yes or no is not acted upon. The client may change their vote from yes at any time (which will redraw the plot on subsequent administrative uses of the 'show' function).

The actual admin and client side code are both very basic, and only exist as a client portal to display and take commands in a while loop

Admin :

```
# Import socket module
import socket
import sys

# Create a socket object
s = socket.socket()

# Define the port on which you want to connect
port = 4444

# connect to the server on local computer
s.connect(('10.0.0.175', port))

#admin receives and prints initial message and menu messages
while True:
    msg = s.recv(1024).decode()
    if not msg == "shutdown":
        print(msg)
        response = input()
        s.send(response.encode())
    else:
        print("Administrator disconnected.  Have a nice day!")
        sys.exit()
```

Client:

```
# Import socket module
import socket
import sys

# Create a socket object
s = socket.socket()

# Define the port on which you want to connect
port = 4444

# connect to the server on local computer
s.connect(('10.0.0.175', port))

print("You are now connected to your classroom server. Just type yes or no into
#Clients don't receive any broadcasts from the server, unfortunately.
while True:
  response = input()
  if response.lower() == 'quit':
      s.send("disconnecting from server.".encode())
      sys.exit()
  else:
      s.send(response.encode())
```

4. Usage:

a. Initial Sign On- When the server is activated, it will confirm the socket creation.

```
C:\Users\LDING\PycharmProjects\Socket\venv\Scripts\python.exe C:/Users/LDING/PycharmProjects/Socket/server_604_v5.py
Socket created...
Socket bound...
Socket is listening...
Attempting Connection.
```

Assuming the administrator signs in first, they will be granted admin privileges.

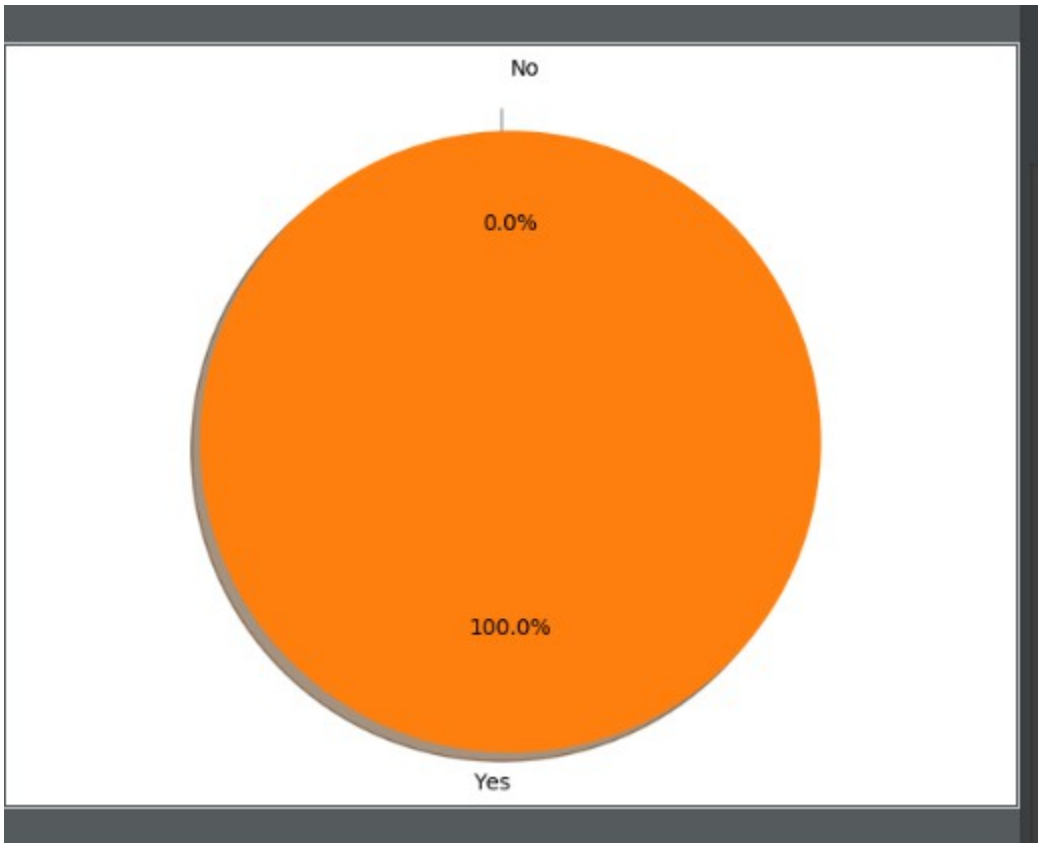
```
C:\Users\LDING\PycharmProjects\socket2\venv\Scripts\python.exe C:/Users/LDING/PycharmProjects/socket2/client.py
Welcome, administrator!
All connected parties will be able to vote on your in class question:
Press any key to continue!
```

Afterwards, all connections are granted student privileges.

```
C:\Users\LDING\PycharmProjects\untitled2\venv\Scripts\python.exe C:/Users/LDING/PycharmProjects/untitled2/venv/Hello.py
You are now connected to your classroom server. Just type yes or no into your input whenever you are asked a question.
```

b. Show-

Users will be able to vote yes and no depending on the instructors' questions. As users type yes and no, the administrator can display the results through the show instruction simply by typing in the command, which will execute a displayable graph on the server side.



c. Clear- Once the administrator is ready to move to the next question, simply typing clear will clear the votes and set up for the next question.

d. Quit – the quit function pulls double duty, ending both the recursive adminmenu function by setting the has_exited variable to True, and signalling to the server that the administrator is prompting a shutdown. On receiving a shutdown

5. Issues and Restrictions

Because this is a framework suitable for demonstrative purposes only, it has several restrictions which would need to be addressed in further work, prior to its adoption for a full-featured application. Each administrator or client is currently handled by initializing a new thread; although this makes asynchronous I/O easy, it has the downside of requiring a very specific order of use cases in order for it to work. Also, because blocking sockets were used, there is no communication from the server to the clients, and therefore no easy way for the server to send the clients a shutdown message in the case of a disconnect by the administrator. This also meant that the idea of having a question input by the administrator and then broadcast to the clients was abandoned, as it resulted in desynchronization issues, wherein one client had not voted on Question #1, but the results had already been cleared and Question #2 was asked. The client then attempted to vote on Question #1, but their vote was added to the results for Question #2, and only then was Question #2 was issued to the client.

Much as with IRC, numerous race conditions arose from the order in which things occurred, resulting in many (many) errors and disconnects. While some efficient error handling and clever code have

eliminated most such errors, a few still exist and can be viewed in the window in which the server is running from time to time. These mostly stem from the admin disconnecting while a client is active, and then the client attempting to vote. Because there is no effective way for the server to close out the clients using the existing code, the clients remain open, even when their sockets are closed by the server. Thus attempting to vote after the administrator has left, even with the server shut down, will result in OS errors related to attempting to "perform a socket based operation on something that is not a socket" (Windows Error 10038). Again, this is an issue for which a solution could be found given a large scale reworking of the existing code.