CPSC 559 - Introduction to Distributed Systems

Group 05 - Final Report

Desmarais, Jesse (00292117)
Gantz, Eric (30031518)
Millard, Joanne (30117800)
Pandey, Pratham (30133275)
Tkachyk, Grant (30077137)
April 09, 2024

Abstract

Our project is Spoofy, a music streaming website inspired by Spotify. Its purpose is to provide users with a reliable and accessible distributed platform for streaming all types of audio content. Spoofy is accessed through a website where users can create a free account. Free accounts can be upgraded to unlock the ability to stream Spoofy's full music library from customizable playlist pages. Website content is managed by administrators using privileged accounts which grant access to interfaces for adding, removing, and editing music-related content, advertisements, and accounts. Client-side content is generated with PHP, and the back end uses the MySQL database management system to store and organize related data. Our system architecture employs a client-server design to enhance efficiency and scalability. Client-to-server, server-to-server, and server-to-database communication are achieved using HTTP requests, TCP sockets, and remote procedure calls respectively. The token ring-based algorithm is used to synchronize our system because of its use in our consistency model and simple implementation. Web server and database processes are replicated to ensure that song data is available for every client no matter what server they connect to. To maintain consistency we utilize the sequential consistency algorithm. Fault tolerance mechanisms for handling omission, crash, and byzantine failure are implemented in our proxy.

Introduction

In today's digital age the consumption of media, particularly music, has undergone a dramatic transformation. With the advent of music streaming platforms such as Spotify and Apple Music, individuals now have unparalleled access to a vast array of audio content at their fingertips. These platforms have not only revolutionized the way we discover, listen to, and share music but have also become integral components of our daily routines and cultural experiences.

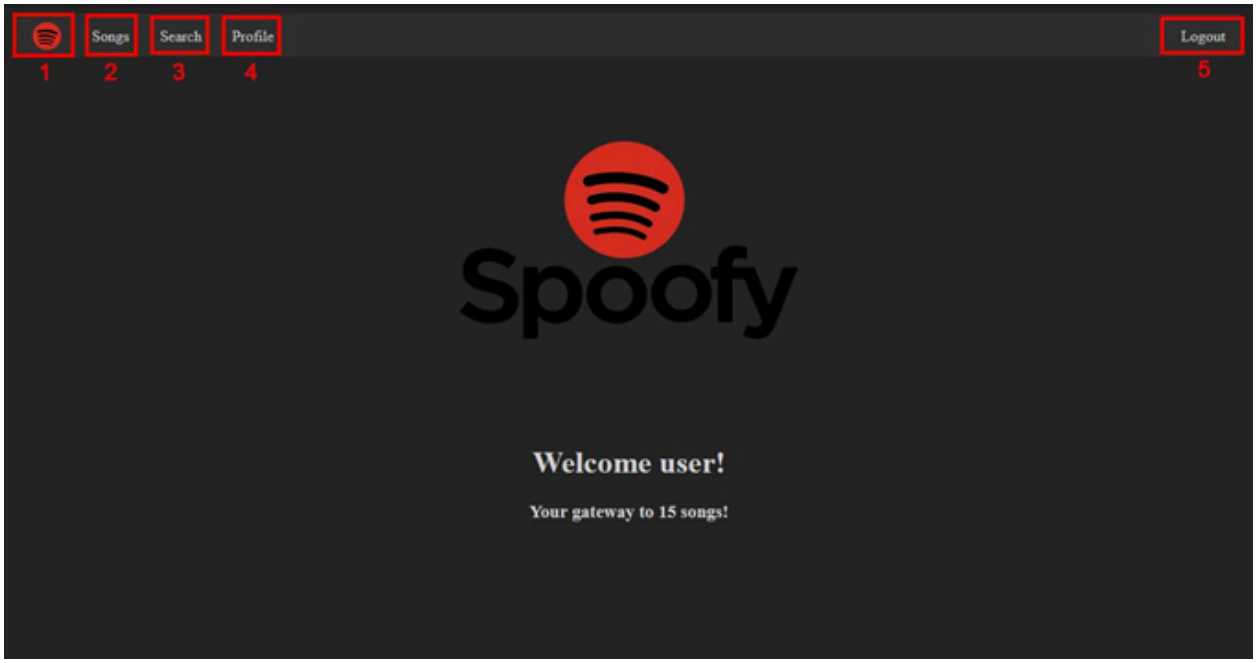
New music streaming platforms have to meet the expectations of users who are accustomed to the high-quality services offered by existing applications. Our project, Spoofy, aspires to carve its niche in the music streaming market by offering users a seamless and immersive experience characterized by reliability and accessibility.

To realize this ambition, Spoofy adopts a distributed system architecture, leveraging cutting-edge technologies and methodologies to optimize performance, scalability, and fault tolerance. This architecture not only ensures the efficient distribution of resources but also enhances the platform's resilience to failures and fluctuations in user demand.

In this report, we delve into the intricacies of Spoofy's distributed implementation, exploring its architecture, communication protocols, synchronization methods, replication strategies, consistency mechanisms, and fault tolerance approaches.

User Guide

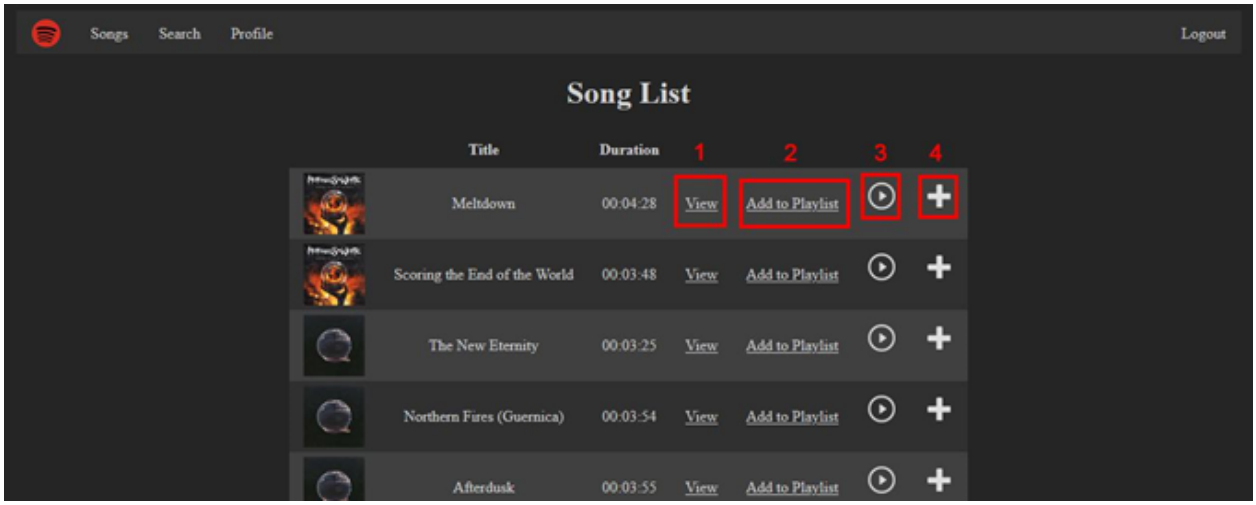
Home Page



The home page for a premium user. Note that this user can view the Song List, instead of Advertisements.

1. Website logo. Click at any point to return to this home page.
2. Navigate to Advertisements.
3. Navigate to Search.
4. Navigate to your account's Profile.
5. Click to log out of your account.

Song List



The songs list page for a premium user. Similar to the song list page for an unregistered user, premium users have the additional functionality of being able to add songs to playlists, play songs, and add them to the queue.

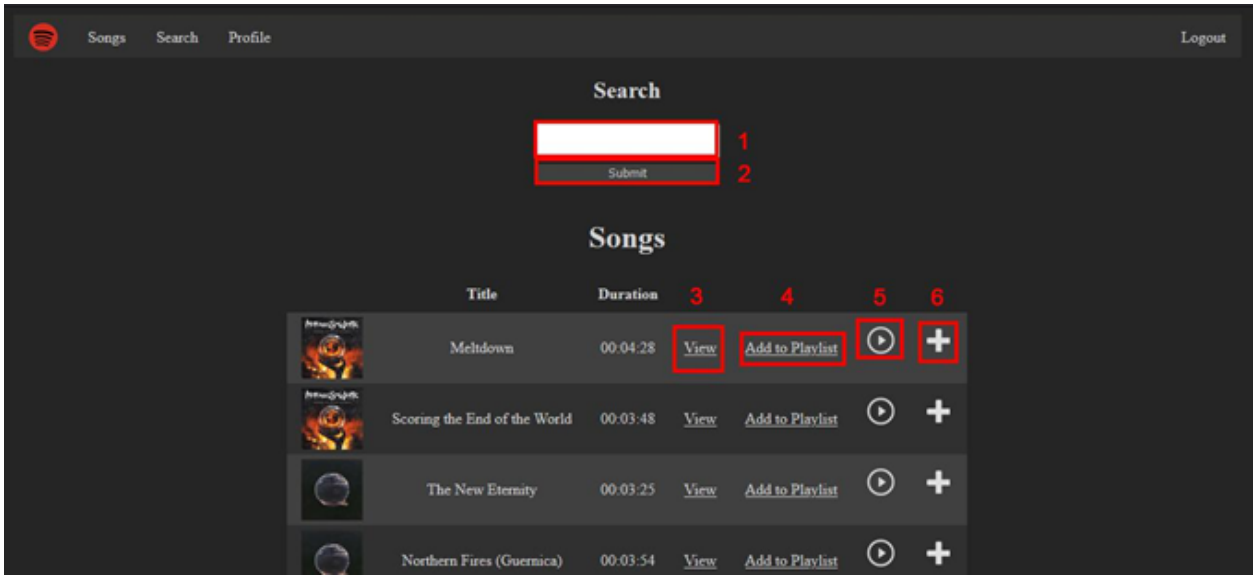
1. Click to view the details of the song.
2. Click to add the song to one of your playlists.
3. Click to play the song.
4. Click to add the song to your queue.

Song Page

A page dedicated to a single song. It shows metadata for that song, such as the title, number of plays, and duration. As a premium user, you are able to play the song, add it to the music queue, or add it to a playlist. You can also enable or disable different music stems that will remain in that state until you log out.

1. The song's title.
2. The song's cover art.
3. Click to play the song.
4. Click to add the song to your queue.
5. Click to add the song to one of your playlists.
6. Click to disable one of the stems of the song.

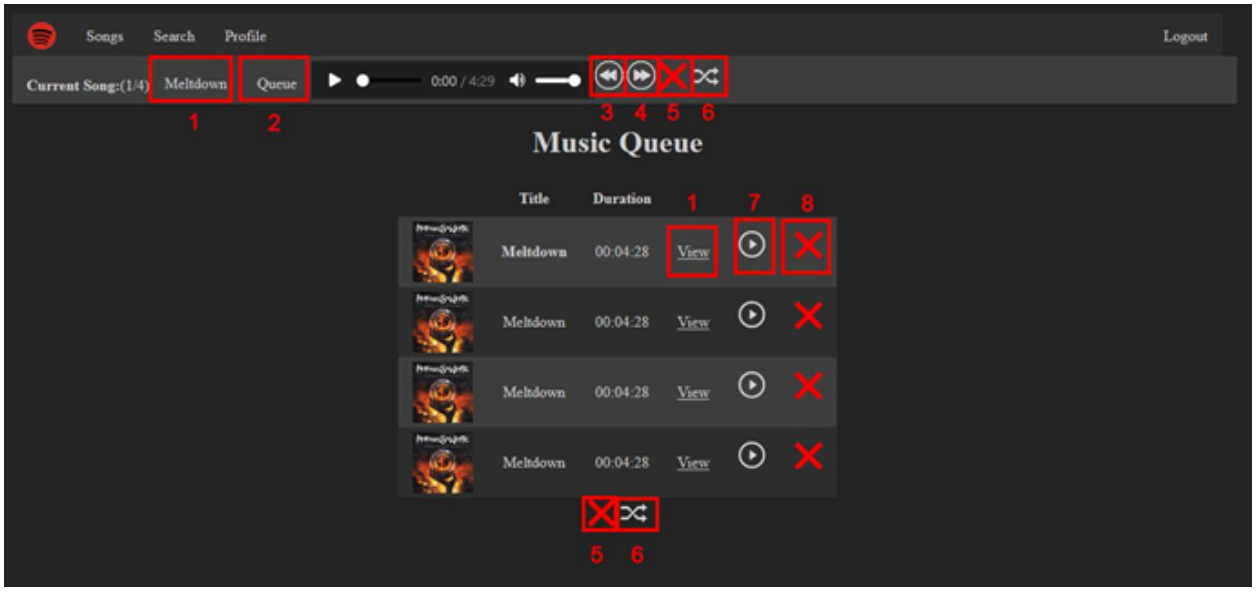
Search Page



The search page of a premium user. Similar to the search page of an unregistered user, it shows Songs, Albums, and Artists that fit the search query. Additional functionality is permitted as you can add songs to playlists and play songs or albums.

1. Click to type in a string to search for.
2. Click to submit your search.
3. Click to view song details.
4. Click to add the song to one of your playlists.
5. Click to play the song.
6. Click to add the song to your queue.

Song Queue



The current song queue. Whenever you play a song or album, or add a song to the queue, it will create a music queue that stores the list of songs you are listening to. You have several options to change the currently playing song, while also being able to shuffle and clear the queue. The queue bar at the top of the screen will persist on any other screen as long as you are logged in and there are songs in the queue.

1. The current song in the queue. Clicking it will navigate to that song's page.
2. Click to view the current queue (the page that is open in the screenshot).
3. Click this button to set the current song in the queue to the previous song, wrapping around to the last song if necessary.
4. Click this button to set the current song in the queue to the next song, wrapping around to the first song if necessary.
5. Click this button to clear the current queue and stop any music from playing.
6. Click this button to shuffle the queue, shuffling the list of songs and setting the current song to the first song of the list.
7. Click this button to play the specified song, setting the current song to the song that is being selected.
8. Click this button to remove the specified song from the queue.

Profile Page

Your profile page as a premium user. Similar to the free user, however, you have the ability to create, edit, and play playlists.

1. Your username.
2. Your account type.
3. The playlist name of a given playlist, stored in the database under this user's ID.
4. The number of songs in the playlist. A playlist is simply a list of songs.
5. Click to go to the page dedicated to that playlist. To view the associated songs and make changes to the list.
6. Click to play this playlist, clearing the current queue and setting the queue to be the list of songs in the playlist.
7. Click to delete this playlist, removing it from your account.
8. The text field to enter the name of a new playlist.
9. Click this button to create the new playlist, under the name specified in 8.
10. Click to log out of your account.
11. Click to cancel your premium membership, reverting your account from premium to free that can only view ads.

Add Song to Playlist

This menu comes up when you choose to add a song to a playlist. It will show all of the playlists attached to your account, and you can select the one to add it to. This will be stored in the database and will persist when you log out.

1. The title of the song that is currently being added to a playlist.
2. The name of a given playlist.
3. The number of songs in a playlist.
4. Click this button to view the playlist before adding the song to it.
5. Click this button to add the song to the chosen playlist.

View Playlist

Viewing a playlist simply displays the list of songs present in it. You have options to play any song present in the playlist or play the entire thing.

1. Click this button to view the song page of a given song.
2. Click this button to remove a song from the playlist.
3. Click this button to play this song, clearing the current queue if there is one.
4. Click this button to add the song to the end of the queue, creating a queue if necessary.
5. Click this button to delete the playlist from your account.
6. Click this button to play the whole playlist, clearing the current queue if there is one.

Architecture

In our system architecture we employ a client-server design, centralizing the interaction between the user interface and the computational services to enhance efficiency and scalability. The client component of our architecture is realized through a web portal which serves as the primary interface for users to interact with our system. This web portal is designed to connect seamlessly to our proxy, which plays a crucial role in managing the flow of requests between the client and the server.

The proxy, functioning as a load balancer, efficiently distributes incoming requests among the available servers. This ensures that no single server becomes a bottleneck, thereby optimizing response times and maximizing the utilization of our server resources. The essence of our service architecture is centralized; when a client makes a request it is processed by the server which then performs the requested service. Upon completion the server communicates back to the client, confirming that the service has been executed and allows the client to consume the service.

Moreover, our architecture is characterized by a multi-tiered approach, where the server not only processes data and executes services but also extends its user interface to the client. This design allows clients to make requests through an intuitive interface while keeping all data manipulation and service execution on the server side. This separation not only simplifies the client-side design but also centralizes the core functionalities within the server, enhancing both security and maintainability of the system. Through this architecture we aim to provide a robust, scalable, and user-friendly platform that efficiently meets the needs of our users.

Communication

We're using 3 types of communication in our distributed system.

Apache HTTP Server

Client to Server - The server communicates with the client via an Apache HTTP web server over a LAN IP address. The web server then receives commands from the client via the proxy and sends back HTML for the client to display based on the received commands.

Then there is a Python script, **`spoofy_distributyor.py`**, listening to the web server for status codes and database queries. If the client makes a change to the website then it calls the **`sendQuery`** function in **`python_connect.php`** to transfer the MySQL statement to the Python script which then performs the database operation. Specifically, the **`sendQuery`** function attempts to create a socket connection with the Python script using localhost and port 9000 to pass the statement to the Python script which will create a thread to handle the statement.

```
// Prepare an insert statement
$sql = "INSERT INTO ALBUM (Title, IsSingle, CoverArt, ReleaseDate, Genre) VALUES ('$title',
$single, '$cover', '$release', '$genre')";
```

```
sendQuery($sql);
header("Refresh:0; url=manage_albums.php");
```

```
<?php

function sendQuery($query){

    $socket = socket_create(AF_INET, SOCK_STREAM, SOL_TCP);
    if ($socket === false) {
        echo "Failed to create socket: " . socket_strerror(socket_last_error()) . "\n";
        exit(1);
    }

    $result = socket_connect($socket, '127.0.0.1', 9000);
    if ($result === false) {
        echo "Failed to connect to server: " . socket_strerror(socket_last_error()) . "\n";
        exit(1);
    }

    socket_write($socket, $query, strlen($query));
    socket_close($socket);
}

?>
```

```
# keep listening for new messages from the Spoofy website and when
# received process the message
(php_socket, addr) = php_listener.accept()

# run the command received from Spoofy
threading.Thread(target=run_cmd, args=(php_socket, out_queue, pool, acks, can_wr, \
                 need_t), daemon=True).start()
```

There is also a socket in the same Python script constantly checking for health checks from the proxy. If an HTTP response code of 400 or less is received then the website is determined to be fine otherwise the proxy drops the server's IP address and attempts to reestablish a connection.

```
# create a listener for proxy health checks
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as proxy_listener:
        proxy_listener.bind(('', PROXY_LISTEN))
        proxy_listener.listen()

        while True:
            client, addr = proxy_listener.accept()

            # while the proxy is still communicating keep getting response codes
            # from the website
```

```
            with client:
                while True:
                    try:
                        data = client.recv(PACKET_SIZE)

                        if not data:
                            break

                        # get a response code from the website and send it back to
                        # the proxy
                        apache_request = requests.head(APACHE_REQUEST)
                        if apache_request.status_code <= 399:
                            client.sendall(b'GOOD')
                    except:
                        break
```

MySQL

Server to Database - When **`spoofy_distributyor.py`** begins execution a pool of connections to the server's instance of the database is populated. Then MySQL statements are sent to the database from the server using a database connection from the pool and the MySQL function **`cursor.execute`**. This is done when a server is in possession of the token and has a write operation to perform or is provided a write operation (see Appendix for description) by a replica with the token. The results of the executed statement are then collected from the server and sent back to the client. All of the messages passed between the web servers and the data servers are through transient and synchronous TCP/IP sockets.

```
# setup a connection to the local copy of the MySQL database
spoofyDB = pool.get_connection()
cursor = spoofyDB.cursor()
```

```
# if the token was received do the following
if token_recv:

    # attempt to make run the received command on the database
    try:
        cursor.execute(mysql_stmnt)
        spoofyDB.commit()
    except mysql.connector.Error:
        spoofyDB.rollback()

    # formulate the write message to send to all the other servers
    write_msg = 'WRITE~' + LOCAL_IP + '~' + mysql_stmnt

    # add the write message to the out queue to send to the other servers
    out_queue.put(write_msg)

    # close connection to Spoofy database
```

```
    cursor.close()
    spoofyDB.close()
```

Python

Server to Server - The Python script **`spoofy_distributor.py`** handles the MySQL insertions, updates, and deletions received from the clients and executes them on a server's local instance of the database.

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as msg_socket:
    msg_socket.settimeout(TIMEOUT)
    try:
        msg_socket.connect((SUCCESSOR, SERVER_SERVER_PORT))
        msg_socket.send(msg.encode())
```

Acknowledgements are also broadcast back to the server which sent out a database write operation to verify that the operation was received and has executed, as well as sending the token in token passing. The MySQL statements are then sent from the Python program to the MySQL database via a transient and synchronous TCP/IP connection where they are executed.

```
# if the message is an ack send it back to the replica that sent
# the database change
if 'ACK~' in msg:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as msg_socket:
        contents = msg.split('~')
        msg_socket.connect((contents[1], SERVER_SERVER_PORT))
        msg_socket.send(msg.encode())
```

Once the database updates are complete the web server will send a **`SELECT`** query to its instance of the database to retrieve the database changes and display them to the user. Below is an example of fetching data about songs in Spoofy. Here it selects the song by song id and determines if the song is set to play via the array key. If so it calls **`play_song`** and **`increment_song_plays`** from **`queue_functions.php`** otherwise it adds songs to the queue with **`add_song_to_queue`**.

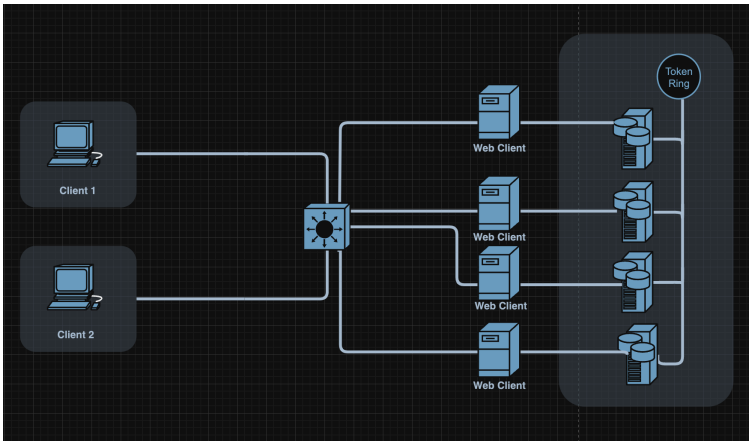
If the Python script makes any changes to the database, such as adding a new song, the client would be able to see it whenever they enter the page.

```
if($_SERVER["REQUEST_METHOD"] == "POST") {
    include "../modules/queue_functions.php";

    // See if a song is being interacted with
    $prepare = mysqli_prepare($con, "SELECT SongID FROM SONG");
    $prepare -> execute();
    $result = $prepare -> get_result();
```

```
    while ($row = mysqli_fetch_array($result)) {
        if (array_key_exists("play".$row["SongID"], $_POST)) {
            play_song($row["SongID"]);
            increment_song_plays($con, $row["SongID"]);
        } else if (array_key_exists("queue".$row["SongID"], $_POST)) {
            add_song_to_queue($row["SongID"]);
        }
    }
```

All 3 types of communication come together in a standard user session in the following ways:

- Initial communication starts between the webserver and the client where user input is forwarded by the proxy/load balancer as an HTTP request to the webserver and then using TCP sockets a MySQL statement is sent to a replica server.
- The server that receives the connection will commit the change to its instance of the database by sending the statement through a TCP/IP connection to MySQL.
- Lastly this server will broadcast the query through a TCP/IP connection to every other server in the ring. Each connection is Transient and synchronous making sure the clients updates are properly received by a server and committed to the database.



Synchronization

To synchronize our distributed system we decided to utilize the token ring-based algorithm. We chose this method because it is a crucial component of the sequential consistency model we used for consistency. Also this method is quite easy to implement, however handling process crashes becomes more difficult and requires us to devise our own methods for dealing with crashes. In the end we ended up utilizing timeouts and socket connection exceptions to register process and replica crashes.

Each server replica must run the **`spoofy_distributor.py`** Python program. This is where the token gets passed amongst the replicas. When the program is being started on all the replicas there will be 1 “primary” replica and the rest are just normal replicas. The only difference between the “primary" and the rest is that the “primary” initiates token passing, so the

program must be started on every other replica before it is started on the “primary”. Once the “primary” replica is started then token passing is initiated in the **`snd_msgs`** function on line 418 of **`spoofy_distributor.py`** by sending a token message directly to the “primary” replica’s successor:

```
if init == '--prim':
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as msg_socket:
        msg_socket.connect((SUCCESSOR, SERVER_SERVER_PORT))
        msg_socket.send(TOKEN_MSG.encode())
```

After this the token is passed amongst replicas by receiving it in a replica’s in queue, keeping it if a write operation needs to be performed on its instance of the database or placing it in the replica’s out queue and sending it to the replica’s successor once a write operation is completed or if the replica does not require the token. The explanation of how the token is utilized for write operations is covered in the synchronization section, so this section will discuss the action of sharing the token between replicas once token passing has commenced and how token passing participates in the detection of failed replicas.

When a token is received from a replica’s predecessor it is placed in the replica’s in queue and is processed in the **`rcv_msg`** function on line 615 of **`spoofy_distributor.py`**. If a replica is waiting to perform a write operation on its instance of the database then it does not pass it along and instead sets a threading event stating that it is allowed to perform write operations. However if the replica does not required the token then it adds it straight back to its out queue to be forwarded to its successor:

```
try:
    rcvd_msg = conn.recv(PACKET_SIZE).decode()

    if rcvd_msg == TOKEN_MSG and need_t.is_set():
        can_wr.set()
    elif rcvd_msg == TOKEN_MSG and not need_t.is_set():
        out_queue.put(rcvd_msg)
    .
    .
    .
except ConnectionResetError:
     pass
```

Once a token is in a replica’s out queue the **`snd_msgs`** function is activated on line 418 of **`spoofy_distributor.py`**. The token is removed from the out queue and sent straight to the replica’s successor:

```
while not out_queue.empty():
    msg = out_queue.get()
    .
    .
    .
    elif TOKEN_MSG in msg:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
             msg_socket:
            msg_socket.settimeout(TIMEOUT)
            try:
                msg_socket.connect((SUCCESSOR, SERVER_SERVER_PORT))
                msg_socket.send(msg.encode())
```

Token passing is a major component in how our distributed system registers failures. Firstly, when a replica attempts to forward a token to its successor a failure can be detected. If a connection attempt is made in order to pass the token to the replica's successor and an exception occurs this likely indicates that the successor has crashed. This scenario is handled in the **`snd_msgs`** function:

```
while not out_queue.empty():

    msg = out_queue.get()
    .
    .
    .
    elif TOKEN_MSG in msg:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
             msg_socket:
            msg_socket.settimeout(TIMEOUT)
            try:
                msg_socket.connect((SUCCESSOR, SERVER_SERVER_PORT))
                msg_socket.send(msg.encode())

                # since the token was passed successfully start a timer
                # now so that it can be checked if a token has been lost
                start_time = time.time()
            except (TimeoutError, ConnectionRefusedError, \
                    ConnectionResetError):

                # set the ip of the replicas that has crashed
                problem_ip = SUCCESSOR

                # remove the successor's ip from the send list
                process_ips(list(SND_LIST), problem_ip, can_wr)

                # set up a drop message to be sent
                out_queue.put('DROP~' + problem_ip)

                # set up token to be passed to the new successor
                out_queue.put(TOKEN_MSG)
```

In this scenario a connection exception has been thrown because of the crashed successor. The program will then register the IP address of the successor/failed replica. The program then recalculates the list of all the replica's IP addresses present in the distributed system and determines the replica's new predecessor and successor server. This is all done in the **`process_ips`** function on line 82 of **`spoofy_distributor.py`**. In this function a list of all the replica IP addresses (including the replica running the function) in the distributed system is compiled and sorted, with the IP address of the failed successor being removed. From this list the replica's predecessor is determined to be the IP address right behind it in the sorted list and its successor is determined to be the IP address directly in front of it in the list. If the replica's IP address is the first in the list then its predecessor is set to be the last IP address in the list and if the replica's IP address is the last in the list its successor is set to be the first IP address in the list. Then the send list of all the other replicas to send messages to in the system is created by removing the replica's IP address from the list. Finally the expected number of acknowledgement messages is calculated based on the number of IP addresses left in the list.

```
# add the local ip to the provided list if it's not already there
if LOCAL_IP not in ip_addrs:
    ip_addrs.append(LOCAL_IP)

# if a crashed replica's IP was provided remove it from the list
if to_remove != '' and to_remove in ip_addrs:
    ip_addrs.remove(to_remove)

# sort the list of replica IPs in ascending order
sorted_ips = sorted(ip_addrs)

# determine where this replica's IP is in the list of IPs
own_index = sorted_ips.index(LOCAL_IP)

# Determine this replica's successor's IP address to send the token to.
if own_index == (len(sorted_ips) - 1):
    SUCCESSOR = sorted_ips[0]
else:
    SUCCESSOR = sorted_ips[own_index + 1]

# Determine this replica's predecessor's IP address.
if own_index == 0:
    PREDECESSOR = sorted_ips[-1]
else:
    PREDECESSOR = sorted_ips[own_index - 1]

# remove this replica's IP from the list, no need to send messages to
# itself
sorted_ips.remove(LOCAL_IP)

# Convert the sorted and formatted list to a deque.
SND_LIST = deque(sorted_ips)
```

```
# Calculate the number of acks this replica should expect back.
NUM_ACKS = len(SND_LIST)
```

Now back to **snd_msgs**. Now that the replica's new predecessor, successor and send list have been calculated it adds a drop message for the failed successor into its out queue. This will now be sent to all the other still active replica's in the distributed system, and once they receive it they will also run the **process_ips** function as well and ensure the failed replica is removed from their send lists. Now that the replica's failed successor has been dealt with token passing can recommence with the replica placing a token message into its out queue so that it can be sent to the replica's new successor.

Another way our distributed system detects failures with token passing is when a replica has not received a token for some time. When a token is sent in the **snd_msgs** function a timer is started. And since the **snd_msgs** function continually runs a while loop for message sending it will keep updating the timer on each iteration of the loop. If the **snd_msgs** function does not see a token for a set amount of time (currently 10 seconds in our implementation) then a timeout will be triggered. The following is a snippet of the pertinent code in **snd_msgs**:

```
# note the start of the send operation to help test for timeouts
start_time = time.time()

while True:

    # note the current time to help test for timeouts
    current_time = time.time()

    # if a timeout has occurred and there are only two replicas in the DS
    # do the following
    if current_time - start_time > TIMEOUT and len(SND_LIST) > 1:

        # determine the IPs of all the crashed replicas
        crashed_replicas = detect_crashes()

        # remove all the failed replica IPs from the send list and
        # recalculate this replica's predecessor and neighbour
        for ip in crashed_replicas:
            process_ips(list(SND_LIST), ip, can_wr)

        # Queue up drop messages to be sent to all the other servers. This
        # can't be done in the above while loop because the send list is
        # constantly being changed so if these message were in the in queue
        # at that time there is a chance an error would occur where they
        # would be sent to crashed replicas. So these messages are added to
        # the out queue now since the send list is finalized.
        for ip in crashed_replicas:
            out_queue.put('DROP~' + ip)
```

```
        # Put a token in the out queue as well so that token passing can
        # commence again.
        out_queue.put(TOKEN_MSG)

    # while there are messages in the out queue do the following
    while not out_queue.empty():
        msg = out_queue.get()
        .
        .
        .
        elif TOKEN_MSG in msg:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
                 msg_socket:
                msg_socket.settimeout(TIMEOUT)
                try:
                    msg_socket.connect((SUCCESSOR, SERVER_SERVER_PORT))
                    msg_socket.send(msg.encode())

                    # since the token was passed successfully start a timer
                    # now so that it can be checked if a token has been
                    # lost
                    start_time = time.time()
```

So in the above code, when the time waited exceeds the timeout amount the waiting replica does not know which other replicas may have failed, just that it has not received a token. So it runs the **`detect_crashes`** function on line 548 of **`spoofy_distributor.py`**. This function sends a connection request to every replica in the system and if the connection request fails then the IP of the replica that failed the connection request is added to a list and returned to the calling code:

```
crashed_replicas = [] # list of replicas that have crashed

# for every ip address in the current list of replica ips do the following
for ip in SND_LIST:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as check_socket:
        check_socket.settimeout(1)
        try:
            check_socket.connect((ip, CHECK_PORT))
        except (ConnectionRefusedError, ConnectionResetError, \
                socket.gaierror, TimeoutError):
            crashed_replicas.append(ip)

return crashed_replicas
```

Back in **`snd_msgs`**, now that the crashed replicas have been determined they are all removed from the replica's send list and a drop message for every failed replica is broadcast to the other active replicas in the distributed system. Finally, token passing is commenced again by the replica that was expecting a token, by it placing a token message in its out queue.

Another way our distributed system detects failures is by waiting for the token in order to process a database write operation. In the **`run_cmd`** function on line 191 of **`spoofy_distributor.py`** the function waits until a threading event is set which states that it can perform write operations on its instance of the database. However this can only happen once the replica receives the token. If the function waits too long for the token a timeout is triggered and the execution of the program continues regardless of whether the token was received or not. The following is a code snippet of what occurs in this instance:

```
# Wait until the token is received to proceed with database changes.
# There's a chance that the token will not come and the wait will
# timeout.
token_recv = can_wr.wait(TIMEOUT)
try_execution = True # indicates that execution should continue, either
                     # the token has been received and the database
                     # update will occur or the token was lost and a
                     # new one will be generated

while try_execution:

    # if the token was received do the following
    if token_recv:
    .
    .
    .
    # if there was a timeout and the token never arrived do the following
    else:
        # if the token never arrived then the predecessor crashed so log
        # its ip
        problem_ip = PREDECESSOR

        # remove the predecessor's ip from the list of ips and recalculate
        # this replica's predecessor and successor
        process_ips(list(SND_LIST), problem_ip, can_wr)

        # set up a message for the predecessor's ip to be dropped from all
        # other replica's send list
        out_queue.put('DROP~' + problem_ip)

        # now that the token is set to be sent again indicate that this
        # replica has the token
        can_wr.set()
        token_recv = True
```

In the above code, if the token was never received and the timeout is triggered the **`else`** block is executed. Since the failure of any other replica is checked for in **`snd_msgs`** then in this instance the replica that failed to pass the token must be this token's predecessor, so the IP of the predecessor is logged. Then the predecessor is removed from the replica's send list and a

drop message for the failed predecessor is sent to every other replica in the system. At this point this replica declares that it has the token and sets its ability to perform write operations on its instance of the database. Execution then proceeds back to the top of the while loop and the write operation is performed on the replica's instance of the database and eventually this replica puts the token back in the out queue and token passing starts again.

The token-ring based algorithm in conjunction with the sequential consistency algorithm ensures that our distributed system stays consistent, and along with our proxy ensures that our distributed system also remains fault tolerant.

Replication

The replicated processes in our distributed system are the web servers and databases. This ensures that data is available for every client no matter what server they are connected to. When a server starts it takes in a list of IP addresses of the other servers and connects to its instance of the database.

```
# Make an ordered list of all other replicase in the DS, determine this
    # replica's predecessor and successor and calculate the number of acks
    # this replica should receive.
    process_ips(sys.argv[3:], '', can_write)

    # create connection pool for local MySQL database
    try:
        spoofyDB_config = {"database": os.getenv('DB_Name'),
                           "user":     os.getenv('DB_User'),
                           "password": os.getenv('DB_Pass'),
                           "host":     LOCALHOST}
        db_pool = mysql.connector.pooling.MySQLConnectionPool(pool_name="pool", \
                                                              **spoofyDB_config)
    except:
        debug_print("Could not create mysql connection.")
        debug_print("Consult README.md for more details.")
        exit(1)
```

Requests from the clients to the servers are handled through the proxy which directs requests to the most appropriate single server. There is no primary server but a group of servers that can be chosen by the proxy. In our project we utilize **`HAProxy`** to direct the HTTP requests from the client to our web servers and act as a load balancer. We can configure our proxy by listing a server with their name and IP address. Usually in **`HAProxy`** to specify how we want to balance our servers by specifying it under backend but if we do not then it defaults to round robin to evenly direct clients to different web servers. Notice we also include the **`cookie`** keyword, this allows the users to stay on the same server even when changing pages with cookie based persistence, ensuring it will continue contacting that server unless it were to go down. The proxy also performs health checks for fault tolerance.

```
backend myservers
  option tcp-check
  tcp-check send PING\r\n
  tcp-check expect string GOOD
  #Format: server <server name> <ip:port> check port <port to send tcp checks to> inter
<check interval> fall <fails before being pulled from rotation> rise <successes to be put
back into rotation> cookie <server name>
  cookie SERVER_USED insert indirect nocache #comment out this line and the 'cookie <server
name>' part of each server line to disable cookies
  #cookies associate a user with the first server they connect to and send all their traffic
to that server so that they stay logged in when they log in.
  server server1 10.0.0.151:80 check port 5150 inter 2s fall 5 rise 5 cookie server1
  server server2 10.0.0.222:80 check port 5150 inter 2s fall 5 rise 5 cookie server2
  server server3 10.0.0.230:80 check port 5150 inter 2s fall 5 rise 5 cookie server3
  #checks handle crash failures by taking a crashed server out of the server pool until it
succeeds health checks
```

Passive replication is utilized to replicate the MySQL database. In our distributed system each server can serve as a primary replica depending on which server the proxy routes the client to. This means that each replica can perform write operations on its instance of the database and broadcast the operation to every other replica to maintain database consistency. Each server replica has its own in queue where messages received from the proxy and other servers are stored in first-in-first-out (FIFO) order.

```
global LOCAL_IP                # local ip address of this replica
out_queue = Queue()            # out queue of messages to send to other
                                   # replicas
in_queue = Queue()             # in queue of messages received from other
                                   # replicas
can_write = threading.Event()  # thread even indicating if this repllica
                                   # can make changes to the databas
need_token = threading.Event() # thread event inidcating if this replica
                                   # wants to make changes to the database
acks = deque([])               # contains acks that a write operation was
                                   # performed across all other replicas
```

Both the in and out queue are created using a Python thread-safe, synchronized queue class called **`queue`**.The out queue stores messages and database statements that need to be sent to other servers. When write messages are created they are labeled as such at the head of the write message string, followed by the IP of the server performing the write, and then the actual MySQL statement. A separate thread is then always checking if the out queue contains messages. If it does it starts sending the out queue contents to all the other replicas in the distributed system.

```
# if the token was received do the following
if token_recv:

    # attempt to make run the received command on the database
    try:
        cursor.execute(mysql_stmnt)
        spoofyDB.commit()
    except mysql.connector.Error:
        spoofyDB.rollback()

    # formulate the write message to send to all the other servers
    write_msg = 'WRITE~' + LOCAL_IP + '~' + mysql_stmnt

    # add the write message to the out queue to send to the other servers
    out_queue.put(write_msg)
```

```
# while there are messages in the out queue do the following
while not out_queue.empty():
    # pull the message at the front of the queue and prepare socket
    msg = out_queue.get()
```

```
# send the message to every other server in the DS
for ip in SND_LIST:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as \
    msg_socket:
        try:
            msg_socket.connect((ip, SERVER_SERVER_PORT))
            msg_socket.send(msg.encode())
            debug_print(f'Message sent to {ip}:\n\"{msg}\"')
        except (ConnectionRefusedError, ConnectionResetError, \
                socket.gaierror, TimeoutError):
            debug_print(f'The server {ip} refused the connection on port’
                        ‘ {SERVER_SERVER_PORT}\n')
```

To ensure that all the messages are received and to keep up consistency, acknowledgements are utilized. Acknowledgements are discussed in more detail in the consistency section.

Consistency

To maintain consistency we decided to utilize the sequential consistency algorithm. We decided to do this because it removes the necessity to use timestamps or track the process of operations for individual replicas, and it ensures that the total order of write operations occurs in the proper order regardless of what replica is performing changes at any given time.

The reading of database information occurs within the website PHP code since the local database of each replica should be up to date with every other replica in the system, so no checks or acknowledgements would have to be made against other replicas to retrieve

information from the database. The following is an example from the file **album.php** where the website is retrieving all the albums from the database:

```
$prepare = mysqli_prepare($con, "SELECT * FROM ALBUM WHERE AlbumID=?");
$prepare -> bind_param("s", $AlbumID);
$prepare -> execute();
$result = $prepare -> get_result();
```

When a replica receives a write (any operation other than a read) operation from a connected client it initiates the **run_cmd** function on line 191 of **spoofy_distributor.py**. For a replica to perform a write operation on its instance of the database it requires the single token that is passed amongst the replicas. To indicate that it requires the token to perform a write, a replica sets two threading events (**threading.Event()**), which are basically thread safe global booleans, to True. First in **run_cmd** the need of the token is set by invoking:

```
need_t.set()
```

This way the replica knows that the next time it receives the token it should retain it to perform the write. Once the need for the token is set the token is listened for within the **rcv_msg** function on line 615 of **spoofy_distributor.py**. Once a token message is received the ability for the replica to perform a write operation on the database is set using the other threading event:

```
rcvd_msg = conn.recv(PACKET_SIZE).decode()
if rcvd_msg == TOKEN_MSG and need_t.is_set():
     can_wr.set()
```

During the time waiting for the token the **run_cmd** function has also been waiting to be able to perform the write operation. Once the ability to write has been set the write operation occurs on the replica's instance of the database:

```
# Wait until the token is received to proceed with database changes.
# There's a chance that the token will not come and the wait will
# timeout.
token_recv = can_wr.wait(TIMEOUT)
try_execution = True # indicates that execution should continue, either
                     # the token has been received and the database
                     # update will occur or the token was lost and a
                     # new one will be generated
while try_execution:
        # if the token was received do the following
        if token_recv:
            # attempt to make run the received command on the database
            try:
```

```
                cursor.execute(mysql_stmnt)
                spoofyDB.commit()
                debug_print('Database update completed\n')
            except mysql.connector.Error:
                debug_print('There was an error updating the database\n')
                spoofyDB.rollback()
```

After the write operation is performed a message containing the MySQL statement is added to the replica's output queue so that the statement can be broadcast to every other replica in the distributed system:

```
write_msg = 'WRITE~' + LOCAL_IP + '~' + mysql_stmnt
out_queue.put(write_msg)
```

The write message is then broadcast to all the other replicas in the distributed system within the **`snd_msgs`** function on line 418 of **`spoofy_distributor.py`**:

```
for ip in SND_LIST:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as msg_socket:
        try:
            msg_socket.connect((ip, SERVER_SERVER_PORT))
            msg_socket.send(msg.encode())
        except (ConnectionRefusedError, ConnectionResetError, \
                socket.gaierror, TimeoutError):
            debug_print(f'The server {ip} refused the connection on port'
                        ' {SERVER_SERVER_PORT}\n')
```

When the other replicas receive the write message they are allowed to perform the write operation on the database immediately since the operation was sent by a replica that is currently in possession of the token. The write message is received by the recipient replicas and placed into their respective in queues in the **`rcv_msg`** function on line 615 of **`spoofy_distributor.py`**:

```
try:
    rcvd_msg = conn.recv(PACKET_SIZE).decode()
    .
    .
    .
    else:
        in_queue.put(rcvd_msg)
except ConnectionResetError:
     pass
```

Since the write message is within every replica's in queue the **`run_remote_cmds`** function on line 673 of **`spoofy_distributor.py`** is activated. This method will remove the write message

from the in queue and perform its operation on each replica's instance of the database. Upon execution of the write operation each replica will put an acknowledgement message in its out queue to be sent back to the replica that currently has the token and initiated the write operation:

```
while not in_queue.empty():

    data_item = in_queue.get().split('~')

    if data_item[0] != '':
        db = pool.get_connection()
        cursor = db.cursor()

        try:
            cursor.execute(data_item[2])
            db.commit()
        except:
            db.rollback()

        ack = 'ACK~' + data_item[1] + '~' + LOCAL_IP + '~' + data_item[2]
        out_queue.put(ack)

        cursor.close()
        db.close()
```

While the other replicas have been performing the write operation the replica with the token has been waiting to receive acknowledgements from all the other replicas in the distributed system in the **`run_cmd`** function:

```
acks_rcvd(out_queue, acks, mysql_stmnt, can_wr)
```

This waiting calls the **`acks_rcvd`** function on line 304 of **`spoofy_distributor.py`**. This function waits to receive all the write operation acknowledgements from all the other replicas in the distributed system by running a while loop. The replica knows how many other replicas there are so there is a specific number of acknowledgements it is expecting before it can break the loop. Once it receives the expected number of acknowledgements the loop breaks. However if another replica fails while the replica is waiting for acknowledgements then a time out will be triggered and the failed replica will be removed from the list of other replicas and the number of expected acknowledgements will be decremented:

```
# retrieve the current time in order to timeout if all acks are not
# received
start  = time.time()

# while the number of acks received doesn't equal the expected number of
# acks, wait
while len(acks) != NUM_ACKS:

    # Calculate how long this loop has been running. If it has timed
    # out determine the replica that has not acked back and proceed to
    # remove it from the list of ips
    current = time.time()
    if current - start > TIMEOUT:

        # convert the acks deque into a list of ips acks have been received
        # from
        ack_ips = ack_ips_to_list(acks)

        # determine the replicas' acks that have not been received from and
        # remove them from the send list and add them to a list of drop
        # messages
        for ip_addr in SND_LIST:
            if ip_addr not in ack_ips:
                process_ips(list(SND_LIST), ip_addr, can_wr)
                drop_msgs.append('DROP~' + ip_addr)

        # add all the drop messages to the out queue
        for drop in drop_msgs:
            out_queue.put(drop)

        # the proper amount of acks have been received so this loop can end
        break

    # if the correct amount of acks have not been received and the timeout
    # has not occurred continue running this loop
    else:
        continue
```

Back in **`run_cmd`** once all the acknowledgements have been received the token will be placed back in the out queue to be broadcast to the replica's successor and the ability for the replica to perform write operations on the database will be removed by setting both the writing and token threading events back to False:

```
# If there are still other servers in the DS then add token to out
# queue and indicate that no writing needs to be done. Otherwise
# if this is the only replica left then it can keep the ability
# to write. But set it so this replica doesn't currently need the
# token.
if len(SND_LIST) != 0:
    out_queue.put(TOKEN_MSG)
```

```
    can_wr.clear()
need_t.clear()
```

By performing write operations in this way we can ensure that they are all performed in the same order across our distributed system so that all instances of the database can remain consistent.

Fault Tolerance

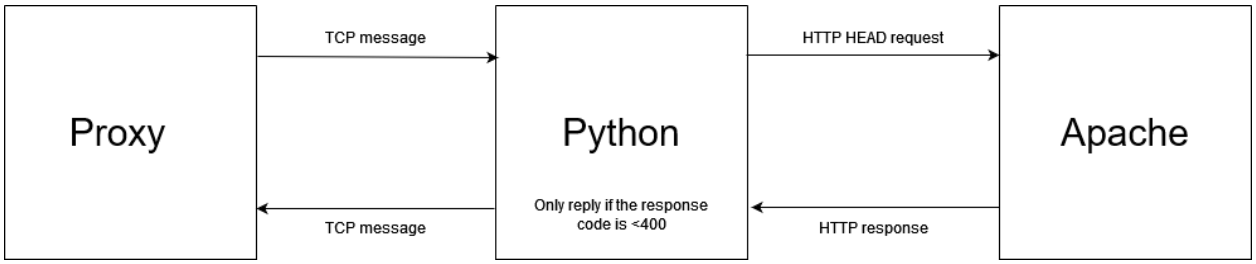
```
defaults
  mode http
  timeout client 10s
  timeout connect 5s
  timeout server 10s
  timeout http-request 10s
  retries 4 #retries sending a message 4 times to handle omission errors

frontend myfrontend
  bind 10.0.0.44:80 #put this machine's ip here
  default_backend myservers

backend myservers
  option tcp-check
  tcp-check send PING\r\n
  tcp-check expect string GOOD
  #Format: server <server name> <ip:port> check port <port to send tcp checks to> inter <check interval> fall <fails before being pulled from rotation> rise <successes to be put back into rotation> cookie <server name>
  cookie SERVER_USED insert indirect nocache #cookies associate a user with the first server they connect to and send all their traffic to that server so that they stay logged in when they log in.
  server server1 10.0.0.151:80 check port 5150 inter 2s fall 5 rise 5 cookie server1
  server server2 10.0.0.222:80 check port 5150 inter 2s fall 5 rise 5 cookie server2
  server server3 10.0.0.230:80 check port 5150 inter 2s fall 5 rise 5 cookie server3
  #checks handle crash failures by taking a crashed server out of the server pool until it succeeds health checks
```

For the three error types, the way we're handling omissions is by resending messages and retrying connections up to four times.

The way we're handling server crashes is through health checks from the proxy. If a server crashes then it will fail health checks and get pulled from rotation, so the proxy will stop sending traffic to it. When a server comes back online it will start passing health checks again and get put back into rotation. Servers that fully crash will have to be manually rebooted and have their database replaced by the current version from another replica.

Byzantine errors are handled similarly, part of the health check is checking HTTP codes, so when a server starts sending incorrect messages it will fail health checks and get pulled from

rotation, then if the problem resolves itself it will start passing health checks and be added back into rotation, if not then it will have to be manually rebooted and updated.



This is the mechanism we're using for health checks: the proxy sends a TCP message to the Python, this is a thread in the same Python script that we're using for consistency, then the Python sends an HTTP head request to the Apache, the apache replies, the Python checks the code and if the code is in the 200-300 range, which means it’s not an error code, then it replies to the proxy.

```
def proxy_health_checker():

        # create a listener for proxy health checks
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as proxy_listener:
        proxy_listener.bind(('', PROXY_LISTEN))
        proxy_listener.listen()

        while True:
                client, addr = proxy_listener.accept()

                # while the proxy is still communicating keep getting response codes
                # from the website
                with client:
                while True:
                        try:
                        data = client.recv(PACKET_SIZE)

                        if not data:
                                break

                        # get a response code from the website and send it back to
                        # the proxy
                        apache_request = requests.head(APACHE_REQUEST)
                        if apache_request.status_code <= 399:
                                client.sendall(b'GOOD')
                        except:
                            break
```

The reason the Python is involved here instead of just sending health checks to the Apache server like you might expect is because if the Python crashes but the server doesn't then that'll cause a bunch of problems for consistency, so we want to make sure the Python is running.

So, with this mechanism the proxy gets a successful health check if and only if the Python is running, the Apache is running, and the Apache isn't sending incorrect responses. If a server fails 5 health checks then it gets pulled from rotation, and if a server that's been pulled passes 5 health checks then it gets added back into rotation.

Example output of the proxy taking servers out of rotation and putting them back in:

```
[WARNING]  (5260) : Server myservers/server1 is UP, reason: Layer7 check passed, code: 0, info: "(tcp-check)", check
duration: 195ms. 1 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.
[WARNING]  (5260) : Server myservers/server4 is UP, reason: Layer7 check passed, code: 0, info: "(tcp-check)", check
duration: 511ms. 2 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.
[WARNING]  (5260) : Server myservers/server5 is UP, reason: Layer7 check passed, code: 0, info: "(tcp-check)", check
duration: 454ms. 3 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.
[WARNING]  (5260) : Server myservers/server2 is UP, reason: Layer7 check passed, code: 0, info: "(tcp-check)", check
duration: 626ms. 4 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.
[WARNING]  (5260) : Server myservers/server3 is UP, reason: Layer7 check passed, code: 0, info: "(tcp-check)", check
duration: 348ms. 5 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.
[WARNING]  (5260) : Server myservers/server1 is DOWN, reason: Layer4 connection problem, info: "Connection refused at
initial connection step of tcp-check", check duration: 46ms. 4 active and 0 backup servers left. 0 sessions active, 0
requeued, 0 remaining in queue.
[WARNING]  (5260) : Server myservers/server5 is DOWN, reason: Layer4 timeout, info: " at initial connection step of
tcp-check", check duration: 2003ms. 3 active and 0 backup servers left. 0 sessions active, 0 requeued, 0 remaining in
queue.
[WARNING]  (5260) : Server myservers/server4 is DOWN, reason: Layer4 timeout, info: " at initial connection step of
tcp-check", check duration: 2003ms. 2 active and 0 backup servers left. 0 sessions active, 0 requeued, 0 remaining in
queue.
[WARNING]  (5260) : Server myservers/server3 is DOWN, reason: Layer4 timeout, info: " at initial connection step of
tcp-check", check duration: 2001ms. 1 active and 0 backup servers left. 0 sessions active, 0 requeued, 0 remaining in
queue.
[WARNING]  (5260) : Server myservers/server2 is DOWN, reason: Layer4 timeout, info: " at initial connection step of
tcp-check", check duration: 2001ms. 0 active and 0 backup servers left. 0 sessions active, 0 requeued, 0 remaining in
queue.
[ALERT]  (5260) : backend 'myservers' has no server available!
[WARNING]  (5260) : Server myservers/server1 is UP, reason: Layer7 check passed, code: 0, info: "(tcp-check)", check
duration: 251ms. 1 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.
[WARNING]  (5260) : Server myservers/server4 is UP, reason: Layer7 check passed, code: 0, info: "(tcp-check)", check
duration: 469ms. 2 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.
[WARNING]  (5260) : Server myservers/server5 is UP, reason: Layer7 check passed, code: 0, info: "(tcp-check)", check
duration: 409ms. 3 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.
[WARNING]  (5260) : Server myservers/server2 is UP, reason: Layer7 check passed, code: 0, info: "(tcp-check)", check
duration: 483ms. 4 active and 0 backup servers online. 0 sessions requeued, 0 total in queue.
```

Summary and Conclusions

The project Spoofy is designed as a music streaming website inspired by Spotify, focusing on providing a reliable, highly accessible, distributed platform for streaming a variety of audio content. It utilizes a free and premium tier system for user access, with additional administrative privileges for system management. The technical architecture is built on a client-server model, employing PHP for client-side content generation, MySQL for data management, and Python for backend distributed functionality, alongside HAProxy for load balancing.

Spoofy's architecture enables efficient request handling through a load balancer, distributing incoming client requests across multiple servers to ensure no single point of failure and optimal

load distribution. The system emphasizes fault tolerance, utilizing health checks to manage server availability in the face of crashes or erroneous behavior, ensuring continuous service availability.

Communication between components is structured around three primary types: HTTP requests from client to server, direct database interactions for data storage and retrieval, and inter-server communication to ensure consistency and replication of database states across the system. This setup supports a scalable, responsive service that can handle varied user requests, from content browsing to interactive functionalities like playlist management and song playback.

To address synchronization and ensure data consistency across the distributed system, Spoofy employs a token ring-based algorithm, facilitating sequential consistency in database operations. This method simplifies the execution order of operations across replicas without requiring complex timestamp management, thereby maintaining a coherent system state.

Replication is achieved through a passive model, wherein each server maintains a local copy of the database, handling client requests individually before propagating changes to other servers. This approach enhances data availability and resilience against server failures.
Spoofy's design and implementation showcase a robust approach to building a distributed music streaming service. By leveraging a mix of established technologies and protocols, it ensures a high level of reliability, scalability, and fault tolerance. The strategic use of load balancing, communication mechanisms, and synchronization algorithms allows for efficient management of client requests and consistent data state across the system, despite the inherent challenges of distributed computing.

The project's architecture supports a seamless user experience, from basic browsing to complex interactions, underpinned by a resilient backend capable of adapting to varying loads and recovering from server failures. Spoofy exemplifies how thoughtful system design, focusing on scalability, consistency, and fault tolerance, can meet the demands of modern web applications.

Appendix

write operation: Any operation performed on an instance of the MySQL database that isn’t a read operation. So INSERT, DELETE, UPDATE, etc.