CS 4624 Final Report:
Crisis Events Webpages Archiving

Matthew Averyt, Jeff Chen, Merna Khamis, Justin Lee, Nihal Mitta

Instructor and Client: Mohamed Farag
Virginia Tech, Blacksburg VA 24061

May 2, 2023

Contents

List of Figures

1 Abstract/Summary

Webpages disappear online rapidly. When something like a crisis event occurs, it is very important to retrieve and preserve all webpages related to that event before they disappear in an effort to record their digital history. Web archiving is a technology that enables storing webpages in a format called WARC (Web ARChive). WARC records save all the pertinent information required to replay the webpage as it was online, such as the HTML data, files, and ads.

The goal of this project is to implement a web archiving system that can archive a large number of web pages depending on user input. To solve this task, we have implemented two main scripts using Python due to its known scripting capabilities, and a user interface to provide a recording and replaying functionality to our system.

The first script's purpose is to go through user-given websites and archive them in the Web ARChive (WARC) format, with one WARC file per website. It is capable of accepting a URL and a collection name to direct the archived URL to, using various Python libraries and packages to do so like pywb and subprocess, as well as waiting between URLs to not overload a server with requests, avoiding being blocked by the target website(s). It operates by reading URLs from a given text file and utilizes multithreading for an overall faster performance during archival.

The second script's purpose is to replay WARC files, showing the archived webpage(s) as it initially was before archival. It is capable of accepting a WARC file (.warc or .warc.gz) inputted by the user to be displayed using the webbrowser library on the user's own web browser. Similar Python libraries are used in this script in its implementation.

The user interface was created using Node.js and React and is based on pywb's WebUI, serving to provide the user with an easier way to use the aforementioned scripts. A Flask script then used to link the UI and scripts for together, allowing for greater usability, with the functions the scripts have to offer to be available for easier use. Using the UI, a user can search for an archived page using its URL and is presented with a pywb calendar to view the website capture(s) in their local browser.

The final deliverables of this project include completed scripts, a user interface, a set of presentation slides, and a final report submitted to our professor and client, Mohamed Farag. The report and presentations show the progress our team made throughout the stages of our project's development.

2 Introduction

2.1 Background

Current solutions lack speed and efficiency, with current tools available for an unknowledgeable public not being fit for mass archival, like Conifer. They can also be unreliable, with some tools being cumbersome to utilize or unable to replay a given WARC file efficiently.

2.2 Objective

The project can be broken down into two aspects: backend and frontend. The objective of the backend team is to be able to take in a large number of URLs and to archive them all into WARC files. The objective of the frontend team is to be able to replay WARC files and make to make use of both recording and replaying user-friendly, as well as providing a user with an easier way to utilize the backend.

2.3 Deliverable

The purpose of this project is to improve upon current solutions on webpage archival. The main deliverables listed below will be presented:

- record.py - A Python script for recording URLs
- replay.py - A Python script replaying WARC files
- A React user interface that links the two scripts together for usability
- flask-app.py - A Python script for defining the Flask App

2.4 Client

Our team's client is our professor Dr. Mohamed Farag, who is currently working at Virginia Tech and is a research associate in the Virginia Tech Transportation Institute's Center of Sustainable Mobility, having research interests in information retrieval, big data, and intelligent transportation systems.

2.5 Team Acknowledgments

Our team is comprised of Matthew Averyt, Jeff Chen, Merna Khamis, Justin Lee, and Nihal Mitta.

3 Requirements

The Crisis Event Webpages Archiving system should fulfill several requirements.

3.1 Functional Requirements

The system needs to meet various functional requirements. It should take a text file with a list of URLs or a WARC file for replay and create a WARC file for each inputted URL. It needs to fetch and archive the specified webpages in the WARC format while preserving all the important information to replay the webpage as it was before archiving. Moreover, the system should be able to replay a given WARC file, showing the archived webpages as they were when archived. It is also important for the system to be efficient and reliable, quickly archiving webpages without overloading the target websites with requests and making sure all necessary components are in the WARC files.

3.2 Non-Functional Requirements

As for non-functional requirements, the system should be user-friendly, allowing even users with little technical knowledge to input URLs and replay WARC files. It must be scalable, and capable of handling more webpages without losing performance or efficiency. Easy maintenance and updates are crucial, with well-documented and organized source code. The system should be compatible with different web browsers and operating systems as well.

4 Design

When designing the system, we decided to use a module-based approach to easily construct components separately before having to bring them together in the final project. The two primary modules of our system are the replay module and the record module. These two modules were linked using a Flask app and API.

4.1 Replay Module

The replay module of the system is focused primarily on the review and replay of any recorded data. This module is able to be used independently of the record module and can replay any WARC file given to it. It can replay any archives that have been generated using the record module, but it can also replay an archive created using a different system.

4.2 Record Module

The record module of the system is primarily focused on the archiving aspect of the system. It consists primarily of two separate Python scripts, one to parse and divide the inputted URLs to be recorded, and another to take that parsed list and record the web pages of each of the given URLs into a separate WARC file. After the processing of all URLs is complete, this record module will route the user into the replay module so that the user can review what they have just recorded.

4.3 Flask App

The Flask app is meant to allow the record and replay modules to communicate effectively with each other, as well as to allow for connection to a user-friendly and understandable UI. Additionally, the use of a Flask App allows us to create our own API which can be used to easily send requests to various modules, allowing for communication between separate programs that allow for modules to be swapped out and advance at different rates. This is intentional to make development easier and also allow future developers to easily swap portions of the program for their own code.

5 Implementation

The project is implemented with a combination of custom Python scripts, JavaScript scripts, HTML scripts, CSS scripts, and the pywb library. The JS, HTML, and CSS scripts are used for pywb's WebUI. replay.py is implemented with a series of subprocess calls which runs external processes with shell commands. These shell commands interact and send data to pywb.

5.1 record.py

The record.py script is designed to create a WARC (Web ARChive) file by recording the content of URLs provided in a text file. The script primarily relies on the pywb Python package to handle the recording process and the requests library to make HTTP requests. The script contains four main functions: archive_url, process_urls, main, and the execution of the main function if the script is run directly. The archive_url function is responsible for archiving a single URL by making a request to the local pywb server with the collection name and URL specified. The process_urls function processes groups of URLs from a queue and calls the archive_url function for each URL in the group. The main function initializes a unique collection name using a timestamp, changes the working directory to the script's directory, initializes the pywb collection, starts the pywb server in record mode, and processes the input URLs by grouping them by domain. A ThreadPoolExecutor is used to run the process_urls function concurrently in multiple threads to improve the performance of the archiving process. There is a maximum of 4 concurrent threads to balance speed and memory usage. Once all URLs have been archived, the pywb server is terminated, and the recorded WARC file is compressed using the warcio command-line tool. The script returns the path to the generated WARC file.

5.2 Flask-app.py

The flask-app.py script is a Flask web application that provides an interface for users to upload URL lists as text files or WARC files, and then archive or replay the content of those files. The application makes use of Flask's routing and request/response handling functionalities along with additional libraries such as flask_cors, werkzeug, and psutil.The "/replaywithurls" route allows users to replay a previously archived WARC file by uploading it, while the "/replay" route is responsible for both uploading a WARC file and extracting its list of URLs to be replayed. The "/upload" route enables users to upload a text file containing a list of URLs, which are then archived and saved as a WARC file. The application also includes helper functions such as allowed_file, is_warc_file, and kill_process_on_port, which respectively check if a file has a specific extension, verify if a file is a WARC file, and terminate a process running on a specified port.

5.3 replay.py

The replay.py script primarily utilizes the pywb Python package to provide functionality for replaying a previously recorded WARC file. Contained in the script are three different methods, main, replay_collection, and replay_warc_file. The main method was written to allow for testing of the replay module directly, and uses the argparse Python library to parse arguments that are passed to the command line when running this script. The main method then calls the other two methods, depending on the arguments passed. The replay_warc_file method is used to directly replay a single WARC file using the pywb replay functionality by taking the path to the file as an argument and using the subprocess Python library to open a new instance of the pywb server. The replay_collection method functions similarly to the replay_warc_file method, but instead of replaying a single WARC file, it instead replays a collection that the user made using pywb. It takes the collection name as an argument, and then uses the Python os library to navigate to that collection before using subprocess to open the collection in pywb so the user can view the contents of the collection.

5.4 App.js

The App component serves as the interface for users to interact with the upload and replay functionality. The handleUpload, handleWarcFileUpload, and handleExpressRoute functions are responsible for uploading text files with URLs, WARC files, and handling the express route, respectively. FileReader is used to read the contents of the uploaded files, and fetch is used to send the files to the server for further processing. The archive_webpage function sends a POST request to the server to replay a given WARC file, while the uploadFileToServer function handles uploading the text file with URLs to the server and receiving the archived WARC file in return. The App component renders a user interface with appropriate input fields, buttons, and event handlers, while also displaying the list of archived URLs upon completion.

6 Testing/Evaluation/Assessment

Testing of the system will primarily consist of two separate tests, the testing of the replay module, and the testing of the record module.

6.1 Replay Module Testing

The testing for the replay module primarily consisted of testing various user interface components, so it was mostly a usability test. We tested to be sure that the UI was easy to use and could be learned very quickly. Initially, our UI was not very usable, leading to multiple iterations of it during our testing process, making it easier to use and more appealing. We also took feedback that we had received after demos we gave during presentations to take outside perspectives into account. This has resulted in the UI that we currently have, but it can still see continued improvement with more iterations.

6.2 Record Module Testing

The testing for the record module was more intensive than the testing for the replay module, as this module relies less on user experience but rather on lengthy computations. The bulk of the testing was done on the actual recording, specifically on ensuring that the recording is done correctly. We tested for correctness in the WARC files that are created by the record module, checking the recorded pages against the webpages that are being recorded to detect any errors in the record process. We also tested for time that recording takes, since a primary concern of the system is speed, and we wanted to ensure that our system not only records information correctly, but also quickly. Multithreading the recording has also been tested using a similar process but has not been extensively tested for performance evaluation.

6.3 Additional Testing

Additional testing will also occur to check other minor components of the system. We will specifically test the linking of the record module to the replay module, as this linking of modules is the most likely point of failure in the system, because it was what was made last, and is also the most difficult to test as the two modules it links are constantly changing. We also wish to test usefulness by comparing our system to preexisting solutions to the problem so that we can prove our system is the best way to record URLs.

7 User's Manual

Our archiving tool is a simple web application that provides a user-friendly UI with the homepage shown in Figure 1. It is designed mainly to archive multiple URLs included in a text file and manage WARC files in an efficient way, and this user manual is going through the features we have in our tool.

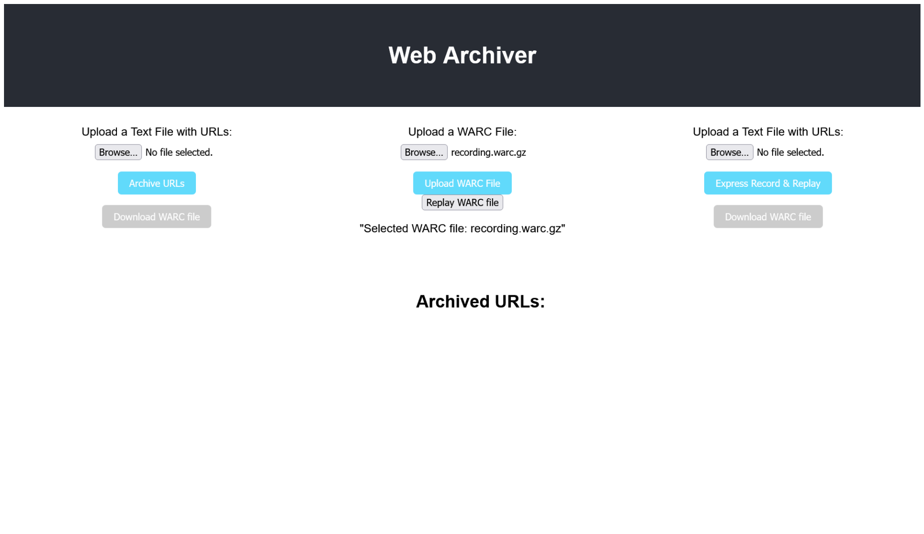

Figure 1: Homepage of the archiving tool.

Archiving Multiple URLs and Downloading corresponding WARC file:

1. To archive multiple URLs at one time, you need to create a text file containing the URLs to be archived, one per line.

2. Navigate to our homepage shown in Figure 1, then click on "Choose File" under the "Upload a text file with URLs" section on the left.

3. Browse your local drive and select the desired text file that is containing the URLs to be archived (in .txt format), then click the "Open" button (Figure 2).

4. Click on the "Archive URLs" button to start the archiving system (Figure 3). A list of the archived URLs will be displayed under the "Archived URLs:" section as shown in figure 3.

5. You can click on any of the URLs to replay it, or simply copy and paste it into your browser to see the live version of the website.

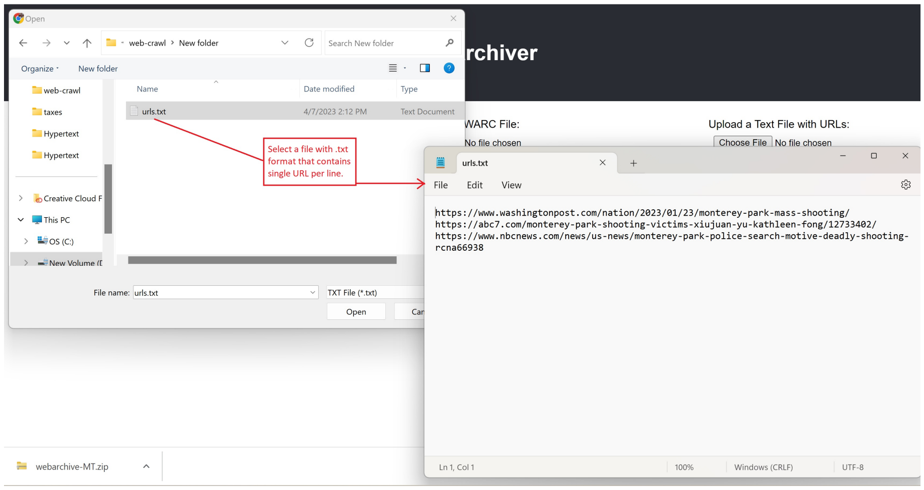

Figure 2: Choosing the text file containing URLs to archive.

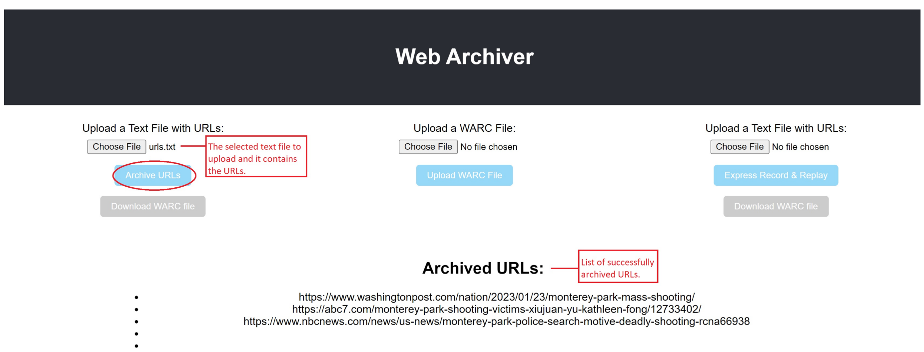

Figure 3: Archive the URLs in the text file and the list of the successful archived URLs.

To Download a corresponding WARC file containing these archived URLs, a second button "Download WARC file" will be activated if the previous process was successful as it is shown in Figure 4. Click it to download the WARC file.

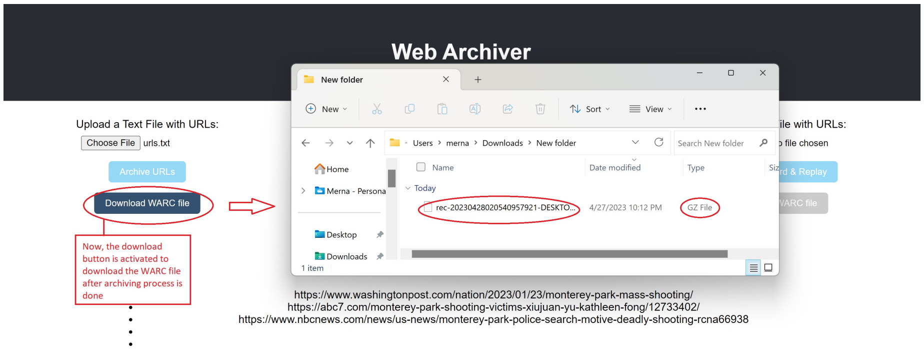

Figure 4: Download the corresponding WARC file after archiving URLs process is done

Upload a WARC File to Replay: Our tool is able to replay a previously created WARC file containing collections in two different ways. To use them, you need to follow the following steps:

1. Click on "Choose File" button and select the desired WARC file.
2. Browse your local drive and select the desired WARC file to upload (can be in .warc or .warc.gz format)
3. If the file in the right format, a new button for replay will show up and the file name(Figure 6).
4. Click on "Replay WARC File". Then, you will be directed to a new page(Figure 7), containing a search bar that you can paste the URL you want to replay, which you can copy from the main page's archived list.
5. When you select the right collection name,
6. Then you will be directed to a new page showing up a calendar(Figure 8), that contains the different captures of the same URL on different days and at different times. You can click on each capture to open a replay version of the URL.

- Another approach to replay a WARC file is to use the last section of our web app which is shown up in Figure 1, to upload the text file containing the URLs and it will directly replay the WARC file without downloading it. This will direct you to the search bar(Figure 7) without any previous steps.

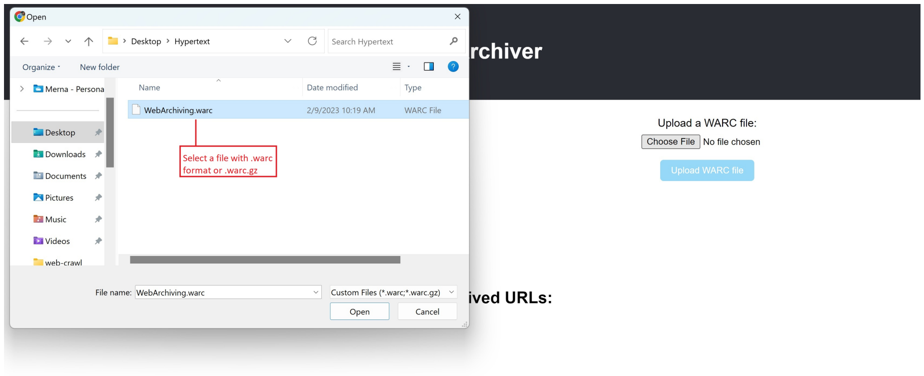

Figure 5: Choose a .warc file format to upload.

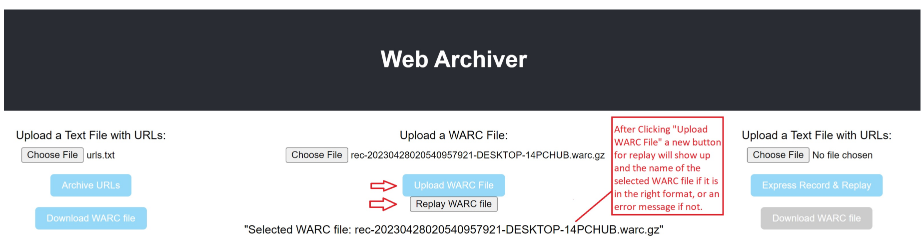

Figure 6: If the WARC file is successfully uploaded, its name will be displayed as shown and new replay button will be available.

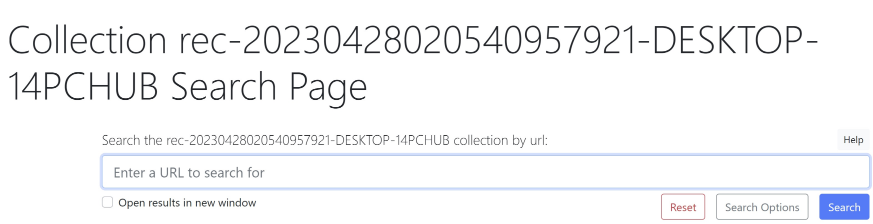

Figure 7: Paste a URL to replay.

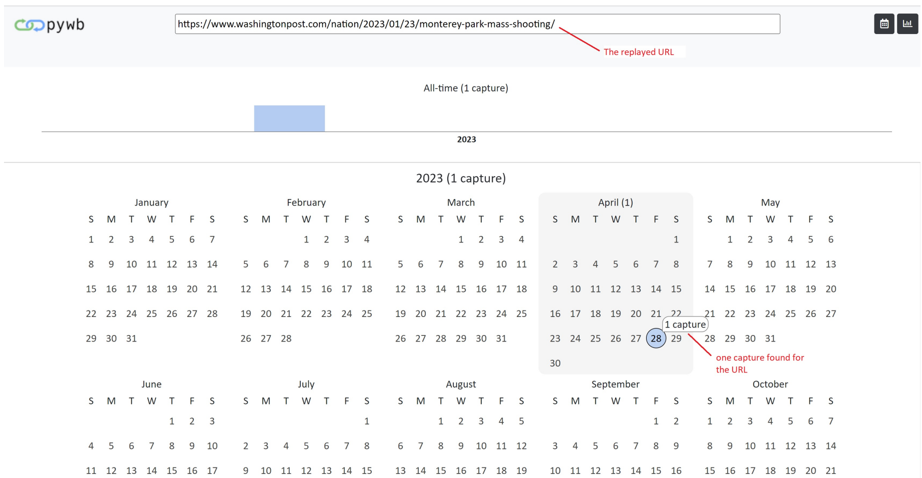

Figure 8: Calendar with different captures for the same URL.

8 Developer's Manual

8.1 Directory Tree

```
web archive
|
- Backend
  |
  - collections
  - src
    |
    - uploads
  - static
  - templates
- Frontend
  |
  - node_modules
  - public
  - src
```

Figure 9: Directory tree of project folders.

At the topmost level, the project consists of 2 folders, Backend and Frontend. The Backend folder contains the main scripts and the following folders: collections, src, static, and templates. The Frontend folder contains the following folders: node_modules, public, and src.

- Backend contains all files required for the backend scripts, the Flask script, and the scripts themselves.
- Collections includes archived WARC files and associated data files.
- src in both the Backend and Frontend contains program files
- uploads folder in Backend contains files uploaded through the front end by the user.
- static contains assets used to build record and replay WebUI.

- templates is an unused folder for storing WebUI templates.
- node_modules and public contains files required to run the front end.

The most important folder to modify program behavior is the src folder.

```
web archive
|
- Backend
  |
  - collections
  - src
    |
    - flask-app.py
    - record.py
    - replay.py
    - shuffleUrls.py
    - warcioreader.py
    - uploads
  - static
  - templates
- Frontend
  |
  - node_modules
  - public
  - src
    |
    - App.js
```

Figure 10: src folder contents.

The Backend src folder contains 3 files: record.py, replay.py, Flask-app.py.

- record.py handles the recording and archival of web pages.
- replay.py handles replaying archived web pages.
- Flask-app.py is an API which allows the front end to connect to the back end.

The Frontend src folder contains 1 file (there may be other files which exist as supplemental files): App.js

- App.js is the React-UI users interact with in order to record and replay WARC files.

8.2 Installation

The following instructions is for Windows OS only.

Software:

- Python 3.9.x
- Node.js
- npm
- Git bash

Installation instructions

1. Git clone https://git.cs.vt.edu/jklee00/webarchive.git
2. Open git bash and enter the following:
   - git checkout MainCode
   - pip install pywb
3. Navigate to "webarchive/Backend/src" and open git bash.
4. Enter into git bash:
   - python -m venv venv
   - "venv/Scripts/activate" with quotations
   - pip install flask
   - pip install psutil
   - pip install flask_cors
   - pip install requests

to run the program, both Front-end and Back-end need to be run concurrently:

- Front-end: navigate to "webarchive/Frontend/src", open git bash, then enter "npm start" without quotations.
- Back-end: navigate to "webarchive/Backend/src", open git bash, enter "venv/Scripts/Activate" with quotations, then "python flask-app" without quotations.

8.3 Program flow

There are 3 ways the program is ran through the Node.js front-end. Since back-end files are designed as individual components, they may be ran separately; however, that is not the main objective of this project.

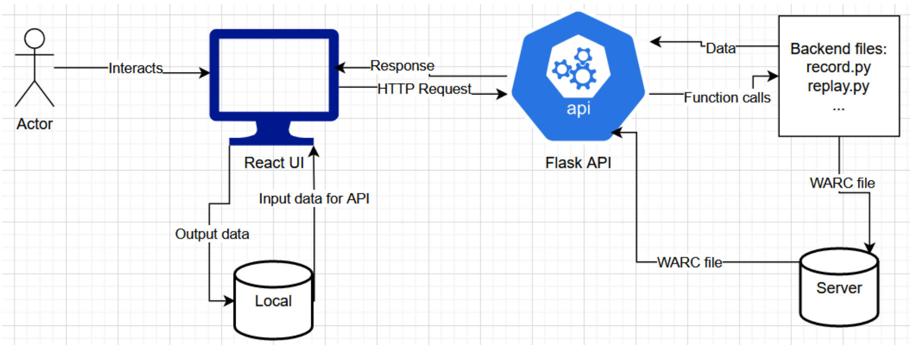

Figure 11: Flow of control.

The user interacts with the React UI, uploading files from local storage as input if needed. From there, the React UI when communicate with the Flask API containing 3 routes: record, replay, and record then replay. The Flask API will communicate to the back-end files to complete it's tasks, referencing and downloading to the server's storage if needed. Once the API's task is complete, it will send a response back to the React UI which includes WARC files and HTTP headers. The React UI will allow the user to download the files if they desire to do so.

8.3.1 Front end: Record

Recording in the React UI involves selecting a text file of URLs, clicking the "Archive URLs" button, and then waiting for the URLs to be archived. When finished, the "Download WARC file" button is enabled and the user may click it to download the WARC file.

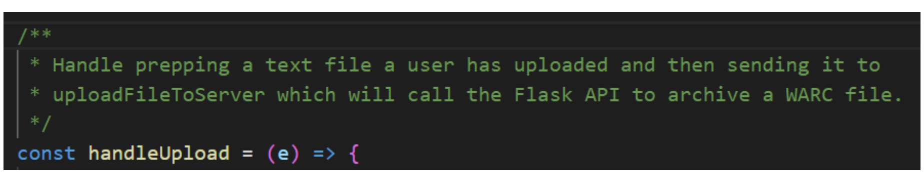

Figure 12: App.js handleUpload function.

```
/**
 * Sends a post request to http://localhost:8000/upload to archive a set of urls. Afterwards,
 * binds WARC to <a> component and enable <a> and <button> component.
 *
 * @param {*} file: Text file containing urls where there is one line per url
 * @param {*} linkid: id of <a> component to bind WARC file to and make unhidden
 * @param {*} btnid: id of <button> component that is wrapped by <a> component referenced
 *                   by linkid.
 * @param {*} replay: true to replay archived WARC file, false to skip
 */
const uploadFileToServer = (file, linkid, btnid, replay) => {
```

Figure 13: App.js uploadFileToServer function.

When a text file containing links has been selected and ”Archive URLs” button is clicked, the React UI calls the handleUpload function. This function grabs the text file the user has uploaded, updates a list shown under ”Archived URLs” in the React UI, and send the text file to the UploadFileToServer function with the id of HTML components to modify and a boolean to determine whether or not to replay the WARC file once processed.

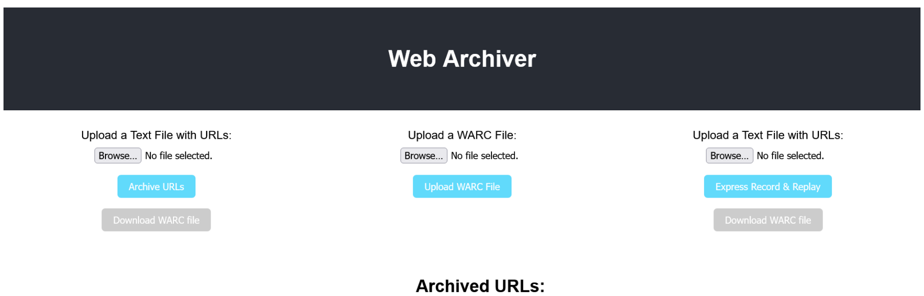

Figure 14: React UI before archival. For the record route, linkid is the $< a >$ component overlaying the ”Download WARC file” button and btnid is the $< button >$ component labeled ”Download WARC file”

Since we are only interested in archiving URLs in this route, the boolean ”replay” is set to False. The components sent is ”linkid” which is the $< a >$ component which will be modified to point to a download link and ”btnid” which is the $< button >$ component which is overlayed by the $< a >$ component. Within uploadFileToServer, a fetch call is prepped to ”https://localhost:8000/upload” which is the Flask API route to record a list of urls into a WARC file. The body of the request is the text file and the request is sent as a POST request.

After the fetch call finishes, the response is handled by saving the file name of the WARC file and the collection name where the WARC file belongs to. In this case, only the WARC file name is needed since the collection name is used only to replay the archived WARC file. The blob which is the WARC file itself is handled by saving the blob to a variable and then creating an ObjectURL for the blob. Then finnaly, the href is handled by linking the linkid component to the blob such that clicking on the linkid component will prompt the user to save the WARc file. The file name to save the WARC file is taken from the previous response handler. The linkid component is then enabled (Normally, $< a >$ components cannot be disabled but a degree of disabling can be done through CSS). The btnid component is enabled, although the btnid component does not have an onclick binding, the $< a >$ element will provide the onclick feature. This is done since $< a >$ components have several unique characteristics which $< button >$ components can't replicate that are essential to being able to download a blob.

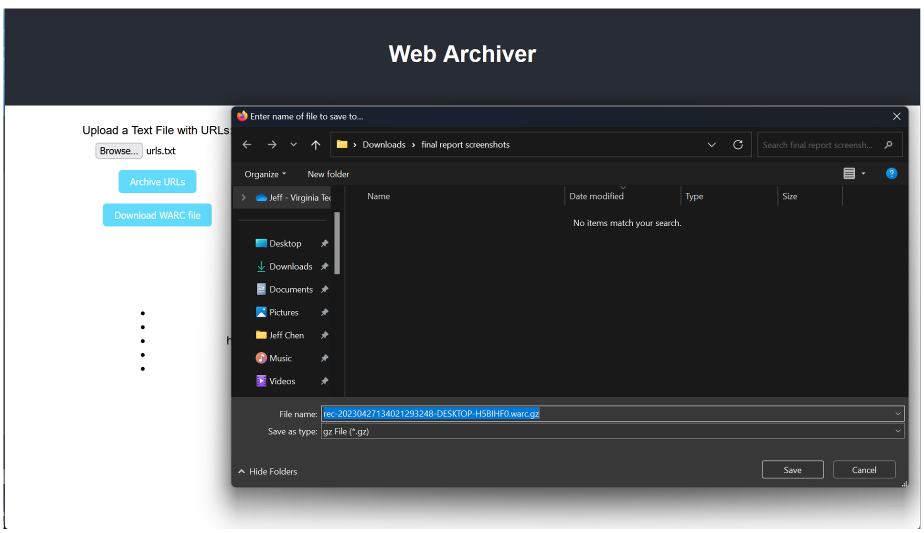

Figure 15: React UI after archival. "Download WARC file" button is now enabled and the overlayed $< a >$ component prompts the user to download the WARC file as pictured.

Afterwards, the user is now able to download the WARC file with the original file name used as seen in the figure above.

8.3.2 Front-end: Replay

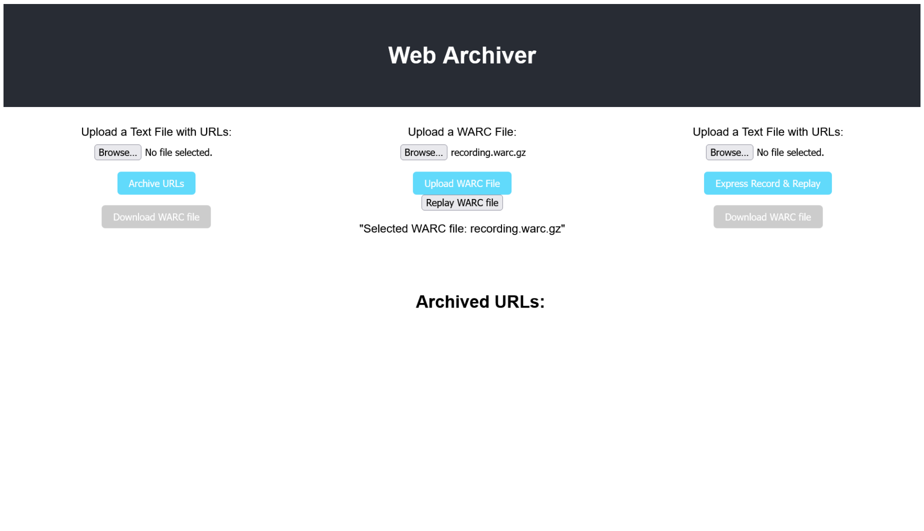

Figure 16: React UI after a file is replayed. The "Replay WARC file" is hidden until the "Upload WARC File" button is pressed.

Replaying in the React UI involves selecting a WARc file to upload. The user will then click "Upload WARC File" to upload it and then "replay WARC file" to replay the file.

```
/**
 * Reveals the "Replay WARC file" button
 */
const handleWarcFileUpload = (e) => {
  e.preventDefault();
  const fileInput = document.getElementById("warc-file-input");
  const file = fileInput.files[0];
  setWarcFile(file);
};
```

Figure 17: App.js handleWarcFileUpload function.

```
/**
 * Preps the WARC file and sends it to archive_webpage to be replayed
 */
const handleReplay = (e) => {
  e.preventDefault();

  const fileInput = document.getElementById("warc-file-input"); //accept the sunmission as file input
  const file = fileInput.files[0];
  archive_webpage(file);
};
```

Figure 18: App.js handleReplay function.

```
/**
 * Sends a POST request to http://localhost:8000/replay to replay WARC file.
 * @param {*} file: WARC file to be replayed
 */
const archive_webpage = (file) => {
```

Figure 19: App.js archive_webpage function.

When the user selects a WARC file to replay and then clicks "Upload WARC File" button, the handleWarcFileUpload function is called. This function reveals the "Replay WARC file" button. When the "Replay WARC file" button is clicked, the handleReplay function is called. This function sends the WARC file to the archive_webpage function. Within the archive_webpage function, a fetch call is prepped to "http://localhost:8000/replay" which is the Flask API route to replay a WARC file. The body of the request is the WARC file and the request is sent as a POST request. After the fetch call finishes, the response is handled by logging the response and the result is handled by logging the result. It is important to note that the React-UI does not open the replay browser, replay.py through the Flask API opens the replay browser.

8.3.3 Front-end: Express

Express in the React UI refers to being able to record and replay with a click of a button. The user uploads a text file of links they want to archive and then click "Express Record & Replay" button. A replay browser is opened and the WARC file is available to be downloaded. Express works similarly as Record except a single variable change resulting in a replaying an existing WARC file instead of having to transfer the archived WARC file to the collections folder.

```
/**
 * Handle prepping a text file a user has uploaded and then sending it to
 * uploadFileToServer which will call the Flask API to archive a WARC file
 * and then replay it.
 */
const handleExpressRoute = (e) => {
```

Figure 20: App.js handleExpressRoute function

```
/**
 * Sends a POST request to http://localhost:8000/replay/<collection_name> to
 * replay an already existing WARC file.
 */
const archive_existing_webpage = (collection) => {
```

Figure 21: App.js archive_existing_webpage function.

The user selects a text file of urls they want archived and clicks "Express Record & Replay" button, calling the handleExpressRoute function. The function body is identical to handleUpload function in recording, except that uploadFileToServer function call has different arguments. Different linkid and btnid is sent which refers to the "Download WARC file" button and its overlay under "Express Record & Replay" button. The replay argument is set to true this time as we would like to replay after archiving the WARC file. The uploadFileToServer call behaves identically to the uploadFileToServer call made for in handleUpload, except that a call to archive_existing_webpages is made at the end of the href handler with the saved WARC file collection name as its argument. The archive_existing_webpage works identically to archive_webpage; however, the fetch call has no body, sends a GET request, and is sent to "https://localhost:8000/replay/<collection_name>". This Flask route replays an already archived collection which means there is no need to include the already archived WARC file in the body of the request to the API.

8.3.4 Back-end: Flask API

The Flask API is how to React UI interacts with the back-end. There are 3 main routes:

1. /replay: replays provided WARC file
2. /replay/<collection_name>: replays existing WARC file using collection name.
3. /upload: Records a set of URLs and returns the WARC file.

The upload routes refer to record and replay routes refer to replay.

POST /replay
The /replay/ route checks if a WARC file was sent in the request body, if none was sent, an error response is returned. Afterwards, the WARC file is sent to an uploads folder. A thread is creating to allow for the replay browser to run continuously and letting a response to be sent asynchronously. The thread's function target is replay.py's replay_warc_file function which takes a path to the WARC file as an argument. The path provided is the WARC file that was sent to the uploads folder. The thread is started and a success response is returned.

GET /replay/<collection_name>
The /replay/<collection_name> route works similarly to the /replay/ route; however, it does not accept a request body and is a GET request. A thread is generated which targets replay.py's replay_collection function with the collection name as its argument. Note that the collection name is provided through the request URL. The thread is started and a success response is returned.

POST /upload

The /upload/ route checks if a text file was sent in the request body, if none was sent, an error response is returned. Afterwards, the file is replayed by saving the WARC file sent to an uploads folder. The WARC file is generated by calling record.py's main function and sending the path to the text file in the uploads folder as its argument. This function returns the complete archived warc file path which is then used to build a response. The response sent contains the archived WARC file as its body, filename as fname header, and collection name as collection_name header. After the response is built, it is sent to the React UI.

8.3.5 Back-end: Record

record.py has 3 functions.

1. archive_url(collection_name, url): records a url to be archived.
2. process_urls(collection_name, domain_queue): process a Queue of URLs provided by domain_queue with the archive_url function.
3. main(url_list): records a list of URLs from url_list into a WARC file.

archive_url(collection_name, url)
To archive a URL, a GET request to pywb's server must be done, which is done in this function.

```
response = requests.get(url_to_archive)
```

Figure 22: Request done to pywb for archival.

process_urls(collection_name, domain_queue)
process_urls calls archive_url for each URL in the lists of URLs. **main(url_list)** main begins by generating a random collection name based on the current date and time as no user input is taken for what to name the collection as. The current directory is changed to the Backend directory as this is where the collections folder exists. The wayback server is opened to generate the collection. After the collection is generated, the wayback server is reopened with several switches to allow for recording URLs into the newly generated collection.

```
subprocess.run([r"wb-manager", "init", collection_name], shell=True, env=env)
```

Figure 23: wb-manager call in record.py generates a specified collection.

```
wayback = subprocess.Popen(["wayback", "--record", "--live", "-a", "--auto-interval", "10"])
```

Figure 24: wayback call in record.py.

Switches used for above wayback call

- −−record which tells the web server to record URLs.
- −−live generates a pseudo collection which points to the live web instead of local WARC files.
- -a enable auto indexing of WARC files into cdjx files.
- −−auto-interval specifies the length of time between auto indexing.

To increase the speed of record functionality, we implemented multithreading. URLs are sorted based on domain names and each new list is added to a queue.

```
domain_groups = defaultdict(list)
with open(uploaded_file_path) as file:
    for line in file:
        parsed_url = urlparse(line.strip())
        domain = parsed_url.netloc
        domain_groups[domain].append(line.strip())

domain_queue = Queue()
for domain, urls in domain_groups.items():
    domain_queue.put((domain, urls))
```

Figure 25: Domain queuing

The threads of our multithreading run the process_urls function. Each process uses takes a queue element as an input, and dequeus an item once it is used. A maximum of 4 threads can be run at once.

After all URLs have been recorded, the wayback server is killed. The archived WARC file path is returned.

8.3.6 Back-end: Replay

replay.py contains 3 functions:

1. main(): allows independent replay from Flask API
2. replay_warc_file(warc_path:str)->None: replays warc file in warc_path

3. replay_collection(collection:str)->None: replays collection corresponding to collection.

main()
The main() has a set of switches to customize behavior:

```
C:\Users\Jeff Chen\Downloads\webarchive\Backend\src>python replay.py -h
usage: replay.py [-h] [-a]

pywb replayer. Replays a stored WARC file or replay previously archived collections through switches.

options:
  -h, --help      show this help message and exit
  -a, --archived  Replay previously archived files
```

Figure 26: Replay.py switches. If no switches are provided, the user is prompted to enter a full path to a WARC file.

If the -a switch is used, replay_collection is called, if no switches are provided, the user is prompted to enter a full path to a WARc file which is used as the argument to call replay_warc_file function.

replay_warc_file(warc_path:str)->None
To replay a WARC file, the wayback server must be opened. This is first done by changing the current directory to Backend directory. A collection name is generated by using the basename of the WARC file provided. The wayback server is called through shell commands which create a collection with a specified collection name and add the WARC file to the collection.

```
subprocess.run(["wb-manager", "init", collection_name])
```

Figure 27: First wb-manager shell command. Generates the collection to store the WARC file.

```
subprocess.run(["wb-manager", "add", collection_name, warc_path], shell=False)
```

Figure 28: Second wb-manager shell command. Adds the WARC file specified at warc_path to the collection.

replay_collection(collection:str)->None
To replay a collection, a wayback server is opened. In a web browser tab, a "http://localhost:8080/*collection*" is opened which points to the wayback server and opens the search window for a the collection.

```
replayer = subprocess.Popen(["wayback", "-a"])
```

Figure 29: Opens a replay browser with auto indexing enabled.

```
webbrowser.open_new_tab(f"http://localhost:8080/{collection}")
```

Figure 30: Opens a specified collection in a web browser tab.

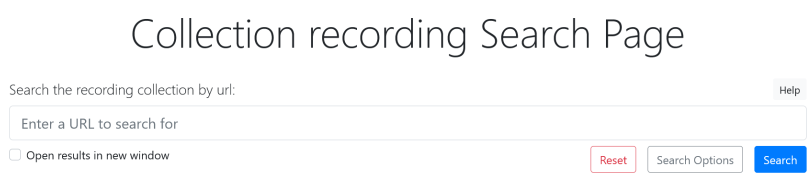

Figure 31: Result of opening a collection in a web browser tab.

9 Lessons Learned

9.1 Timeline

Milestone	Description	Date
Project launch	Project launches and team is finalized.	1/31/2023
Finalized project requirements	Project requirements finalized	1/31/2023
Initial Project Presentation	Presented information about the project and requirements to gather feedback.	2/16/2023
Single URL recording	Developed a Python script for recording a single URL	2/23/2023
Replay WARC collections	Developed a Python script that replays collections in the project directory or replays a locally stored WARC file. Launches pywb WebUI in browser.	2/23/2023
Presentation 1	Finished and finalized presentation 1.	2/27/2023
Finalized program architecture	Finalized program architecture as well as determined which project components needed to be worked on.	2/28/2023
WARC file scanner	Developed a Python script which examines the contents of a WARC file to obtain user input URLs.	3/14/2023
Single URL recording refinement.	Improved single URL recorder script in preparation for bulk URL recorder.	3/17/2023
WebUI extension	Work in progress extension to merge WARC scanner functionality with the front end.	3/18/2023
Interim report	Finished final report draft about the project and its current progress.	3/20/2023
Presentation 2	Finished and finalized presentation 2.	3/27/2023
URL Scrambler	Added URL scrambler pre-process script.	3/27/2023
Multiple URL Support	Added multiple url, single threaded execution to recording WARc files.	4/04/2023
Connected Recording to Frontend	Frontend connected to Record.py through Flask API.	4/09/2023
OS Compatibility	Confirmed cross-platform compatibility.	4/13/2023
Connected Replay to Frontend	Frontend connected to Replay.py through Flask API.	4/09/2023
New UI	Created a new bootstrap UI for to be used as our front end.	4/16/2023
Final Presentation	Finished and finalized the final presentation.	4/17/2023
Final Report 1st Submission	Submit 1st submission to VTechWorks.	4/27/2023
Finalize Project	Work on the project halted and finalized for the semester.	4/29/2023
Final Report Final Submission	Submit final revision of report to VTechWorks.	5/03/2023

Figure 32: Timeline from the beginning of the project up to now.

9.2 Problems

Several problems were encountered during the development of this project.

- The first major issue was learning how to use pywb, the main library used to record and replay files.
- The second major issue was delegating roles which is related to the previously mentioned issue.
- The third major issue is involved with extending the current functionality of pywb, mainly concerning the WebUI.
- The fourth major issue is implementing a URL Scrambler. The URL scrambler would ideally process a list of URLs in a way such that the maximum number of threads matches the number of domains within the url list. However, this approach is challenging to implement.
- The fifth major issue is configuring the program to be Multi-threaded. This step relies on a working pre-processor step described in the previous item.
- The sixth major issue is implementing Flask as an API for the back-end to be called from the front-end. None of our members are familiar with Flask and using an HTTP API before.
- The seventh major issue is ensuring that collaborative work is completed on schedule.
- The last major issue is installing packages the Python program relies on.

9.3 Solutions

- The first issue involving unfamiliarity with pywb was solved mainly through reading through documentation and experimenting with the library. One aspect which couldn't be resolved through reading documentation was invoking pywb's Vue UI based WebUI. This feature was released in the latest major pywb release (2.7) which came a few months before the beginning of this project. As a result, documentation on how to invoke pywb's Vue UI was sparse. After much investigation, it was discovered that pywb had to be built locally and the static folder within the built files was responsible for the Vue UI. Transporting the static folder to where the WARC collections were stored finally allowed pywb's Vue UI to be used.
- The second issue concerning delegation of roles was solved through gaining proficiency of pywb such that roles could be delagated fairly in a way which also involved development with pywb.

- The third issue concerning extending pywb's functionality is still a work in progress but much work has been done. Pywb does not support a way to view all user input URLs within an archived WARC file, leading to poor user experience. To solve this issue, we would need a thorough understanding of WARC files and a way to modify pywb's WebUI. To gain an understanding of WARC files, we consulting a document on the WARC ISO. To distinguish between user input URL records and non user input URL records within a WARC file, comparisons were made between record headers to find a header which is shared by many user inputted URLs and none of the non user inputted URLs. Despite having a solution to identify user input URLs, there is a need to improve the time efficiency of the solution. Our current solution examines each record sequentially. Although this solution works for small WARC files, WARC files often contain thousands of records which may take minutes to process when a WARC file is sufficiently large enough. For modifying pywb's WebUI, we have located the specific pywb files that must be modified but work to modify the WebUI is still in progress.

- The fourth issue concerning URL scrambling was never fully solved, since we decided that the other issues faced were more pressing, and that URL scrambling and multi-threading were less concerning. We briefly had a method that would scramble URLs in a way that would suit a single thread, simply by ensuring that no consecutive URLs had the same domain name. However, this was never properly tested, and did not make it into the code that exists now.

- The fifth issue concerning multi-threading was also not ever fully solved. Once we realized how slow pywb recording was, we immediately recognized that multi-threading would allow us to complete the record process much quicker, but considering the record script had to be completed before multi-threading could be implemented, the issue was pushed further and further back. We did eventually implement multi-threading, so there is now a substantial improvement in record speeds, but it is not as fully implemented as we wish it could have been, and lacks the URL shuffling, so may be susceptible to blocking from domains that receive multiple requests back-to-back.

- The sixth issue concerning the Flask API was solved late in the development process, but solving it was very important to our project's development. Our front-end team worked to build it in a manner such that it would act solely as a bridge between the front and back ends, ensuring that any changes to the code (excluding the change of method names) on either end would not require any retooling of the script. There were some issues that occurred once the implementation was complete, since the front-end team was developing on Windows machines, but the back-end team was developing on Apple machines. This was easily resolved, as the ports chosen to be used by the Flask API were reserved ports on

Apple machines, so we simply had to change the port numbers for the problem to be fully resolved.

- The seventh major concerning the work being finished on time was a problem that we struggled with throughout the entire process and never fully resolved. As is typical with a project of this size, many issues occurred during development, and the group thought that the issues could be resolved quicker than they could actually be resolved. Because of this, some delays naturally arose, as the group worked to get better at time estimation. Additionally, all group members had periods of time where they had other concerns, either from other classes or from outside school. This also caused delays, but were something that could be worked around, as for the most part they were expected issues and could be planned around. The problem as a whole was never fully solved, but by working together as a group and communicating, we were able to reduce the problem to a manageably small issue.

- The last issue concerning the installation of Python packages was solved through trial and error, but was made all the more difficult because of the different operating systems that were used by different group members. The issue mostly arose because the most recent versions of pywb and Flask had an overlap of dependencies, but needed different versions of these dependencies. This was resolved relatively easily by the Windows users, since they simply created a Python virtual environment and installed Flask there, while having pywb installed normally. Because of the virtual environment, they were able to avoid the issues of differing versions and were able to overcome the problem simply. This was more difficult for the Apple users, since the Mac OS does not work well with the Python virtual environment. Because of this, the Apple users had to install an earlier version of Flask and its dependencies to account for the co-dependencies. The earlier versions of Flask and its dependencies were found through trial and error, but were found eventually and the issue was resolved fully.

9.4 Future work

There are several items we wish to cover for the rest of this semester before examining stretch goals if time permits.

1. Extending pywb WebUI to display a list of links to user input URLs, significantly improving user experience with replaying archive data.

2. Bulk URL recording and workaround for rate limiting, this would significant reduce the amount of time and work needed to record a set of URLs.

3. Fully testing the current multi-threaded URL recording implementation for ensuring improved performance.

4. Having a system that guarantees URLs are fully shuffled before recording to prevent getting blocked by a website.

5. Adding a way to view already archived collections without recording new URLs or viewing a WARC file that is not within the collections.

6. Having a better user interface to make it more reliable and more user-friendly.

7. Connecting the back-end with the front-end in a more efficient way.

10 Acknowledgements

Our client: Dr. Mohamed Farag - mmagdy@vt.edu

We would thank our client, Dr. Farag, for allowing us to work on this project and working with us by giving us advice and guiding us through the entire process.

11 References

[1] "Webrecorder website," Webrecorder website, 2023. [Online].
Available: https://webrecorder.net/. [Accessed 21-Mar-2023].

[2] "Selenium website," Selenium website, 2023. [Online].
Available: https://www.selenium.dev/. [Accessed 21-Mar-2023]

[3] "Pywb documentation," Pywb documentation, 2023. [Online].
Available: https://pywb.readthedocs.io/en/latest/. [Accessed 21-Mar-2023]