File Uploads

HTML Forms - POST

- Specify multipart encoding to receive each input separately in the body

```
<form action="/form-path" method="post" enctype="multipart/form-data">
    <label for="form-name">Enter your name: </label><br/>
    <input id="form-name" type="text" name="commenter"><br/>

    <label for="form-comment">Comment: </label><br/>
    <input id="form-comment" type="text" name="comment"><br/>

    <input type="submit" value="Submit">
</form>
```

HTML Forms - POST

- Content-Type specifies a string that separates each input
- Each input has its own headers
- Great for submitting different types of data in the same form
    - Required for file uploads

```
POST /form-path HTTP/1.1
Content-Length: 252
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryfkz9sCA6fR3CAHN4

------WebKitFormBoundaryfkz9sCA6fR3CAHN4
Content-Disposition: form-data; name="commenter"

Jesse
------WebKitFormBoundaryfkz9sCA6fR3CAHN4
Content-Disposition: form-data; name="comment"

Good morning!
------WebKitFormBoundaryfkz9sCA6fR3CAHN4--
```

File Uploads

- We have to use multipart/form-data to upload files
  - If not, browser only sends the filename
- Add an input with type "file"
  - The browser does the rest
  - Users will be able to choose a file to send

```
<form action="/file-upload" method="post" enctype="multipart/form-data">
    <label for="form-name">Enter your name: </label>
    <input id="form-name" type="text" name="commenter">

    <label for="form-file">File: </label>
    <input id="form-file" type="file" name="upload">

    <input type="submit" value="Submit">
</form>
```

File Uploads

- When our server receives the file it will appear in one of the parts of the multi-part POST request
- The content type will tell us the type of file
- The body of the part will contain all the byte of that file
    - Can write these bytes to a new file on our server to save that file

------WebKitFormBoundarygVWEOc5JIyJ1qthO
Content-Disposition: form-data; name="commenter"

Jesse
------WebKitFormBoundarygVWEOc5JIyJ1qthO
Content-Disposition: form-data; name="upload"; filename="discord2.png"
Content-Type: image/png

<bytes_of_the_file>

File Uploads

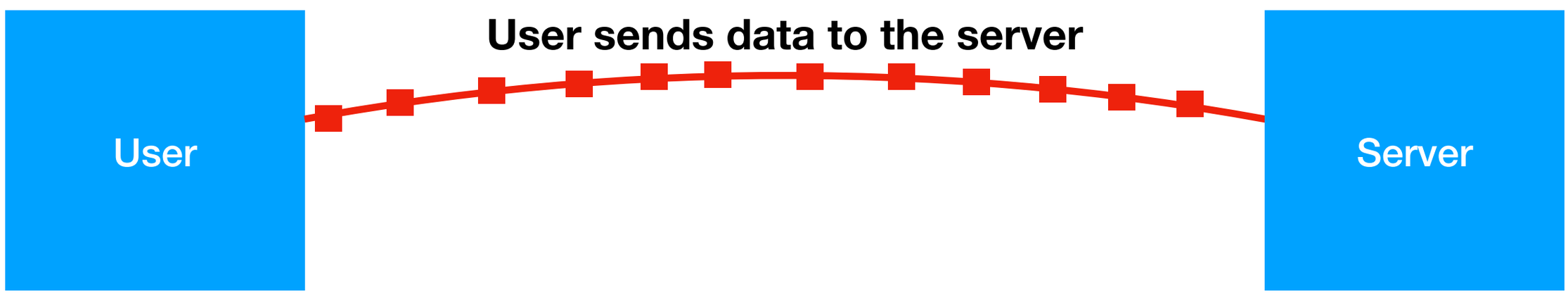
- When receiving the bytes of a file, do not apply any encodings
  - When we received bytes representing text we decoded it by interpreting the bytes as UTF-8 encoded text
  - When receiving files we are often interested in the raw bytes (Ex. Images, videos)
  - The files will be encoded with algorithms other than UTF-8 (Ex. png, mp4)
- Attempting to treat a binary file as a UTF-8 String will only cause bugs and headaches

Buffers

- TCP socket libraries will use buffers
- No matter your language/library you will have a method/ function that reads bytes from the socket
    - Called when there are bytes that arrive over the socket
    - Returns some bytes of the request

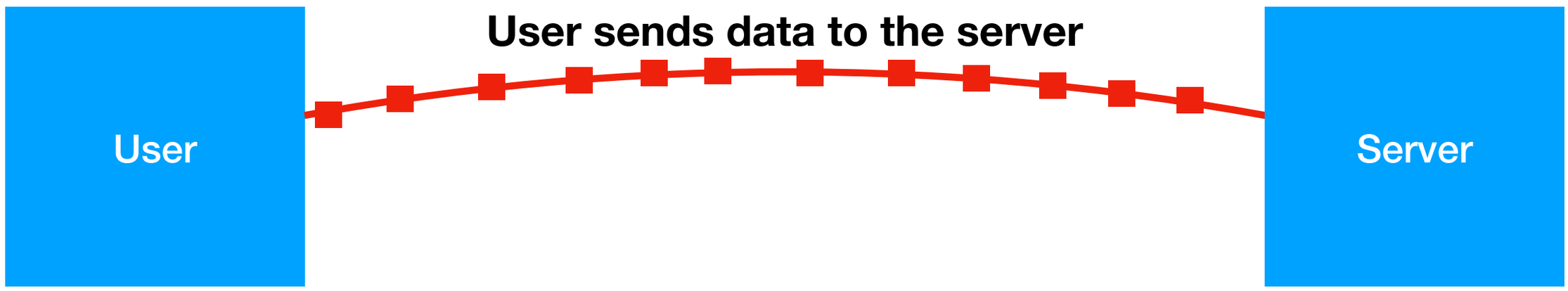
Buffer Questions

- What happens when the user has a lot of data to send?
- What if the user has a slow connection?
- Does the socket server wait for all of the data to be received before calling your code?
- What if the data takes an hour to send?
- What if the data contains streaming video that never ends?



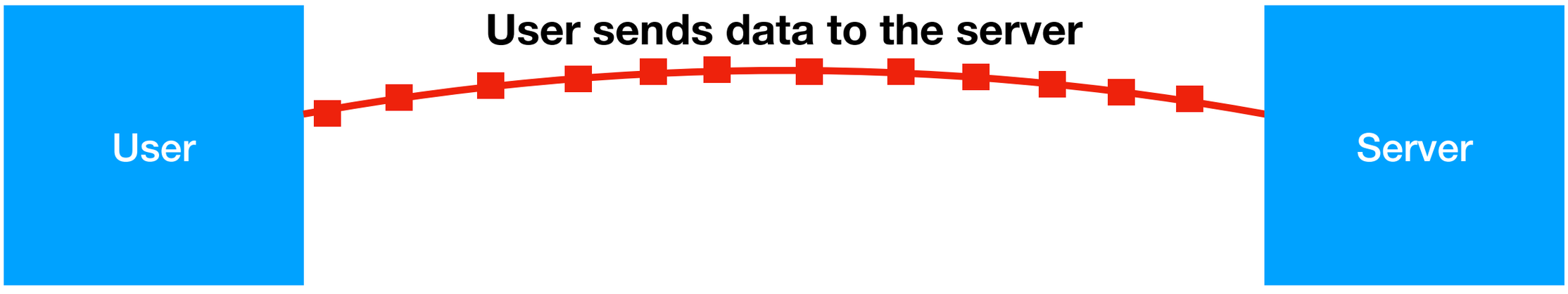
Buffer Answer

- The socket calls your code when there is data to read even if it's not the entire request
- The socket server will have a buffer size, typically a few kB, and will read **at most** that many bytes in a single call
    - For GET requests the entire request is smaller than the buffer



Buffers

- Now that we're handling file uploads we must be aware of these buffers
- The server will need data that persists across multiple calls that read bytes from a socket
  - Create data structures that store the bytes read from a request
  - Combine the bytes from multiple calls to receive the entire file



Buffers

- When receiving a large request
  - Read bytes from the socket
  - Parse the headers
    - Find the Content-Length header and store this value
    - Keep reading bytes from the stream until you have Content-Length number of bytes in your data structure
    - Process the request

We Now Support File Uploads

But What Can We Do With Them?

Hosting Files

- Let's make all uploaded files available to our users
- Users can make a request for a file by it's name
- We need to create a url scheme to accomplish this
    - Ex. GET /image/cool-picture.png
    - Ex. GET /image?filename=cool-picture.png
- More string parsing to find which file to serve
    - If the path starts with /image, read the filename and return it
        - Or a 404 if the file doesn't exist on the server

Hosting Files - Security

- You'll have code that effectively does:
  - Receive request for /image/<filename>
  - Parse the path and extract filename
  - Read the byte of the file
  - Send those bytes in the response

- ... and someone makes the following request:
  - GET /image/~/.ssh/id_rsa

Hosting Files - Security

- GET /image/~/.ssh/id_rsa
  - This attacker now has your private encryption key

- First line of defense:
  - Remove all "." and "/" characters
  - Add logic to ensure user can't access files outside the public directories