INDIAN INSTITUTE OF TECHNOLOGY MADRAS
सिद्धिर्भवति कर्मजा

IIT Madras
ONLINE DEGREE

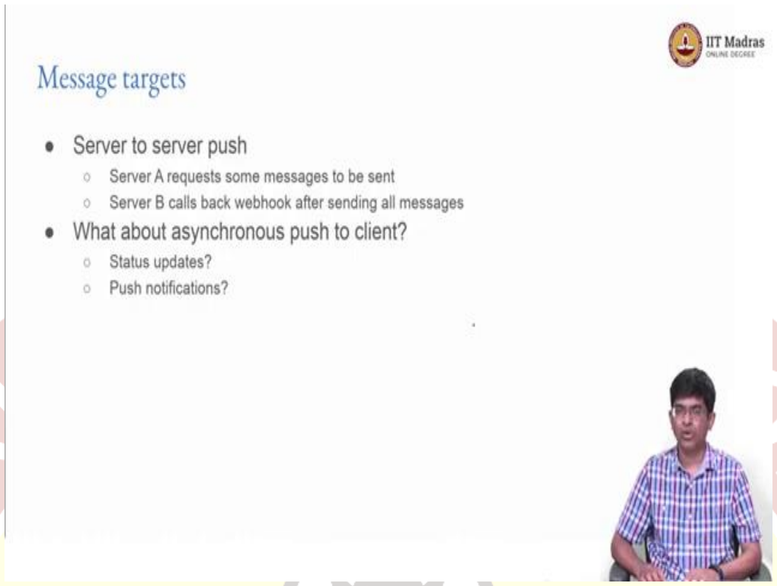
Programming and Data Science
Professor. Nitin Charachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras
Modern Application Development – 2
Push to Client

Hello, everyone, welcome to Modern Application Development, part 2. So far we have been looking at different ways of doing sort of what we were calling server to server communication, we had two servers, A and B that are each providing some kind of service. And let us say that B needs to alert A that it has either completed a task or something that needs to pass a message back to A web hooks turned out to be a nice simple way of providing it without requiring a full blown message hook, a messaging queue.

What if instead, I want to get the so called push notifications, I want something to pop up on my screen on my browser, or on my phone, that tells me that there is actually something that I need to go and check a message for. And this we are used to all the time, whenever there is an SMS message, something pops up on our phone. And now whenever we have WhatsApp or email or any of these other messaging systems, we like to see something that tells us yes, there is a new message that comes along, that has come along and maybe you should go take a look at it.

Of course, we have now reached a stage where probably there are too many of such messages. So, maybe this pushing to client is not a good idea, after all. But that is more of a philosophical discussion, what we would like to see is how do we solve it from a technological point of view.

(Refer Slide Time: 01:29)

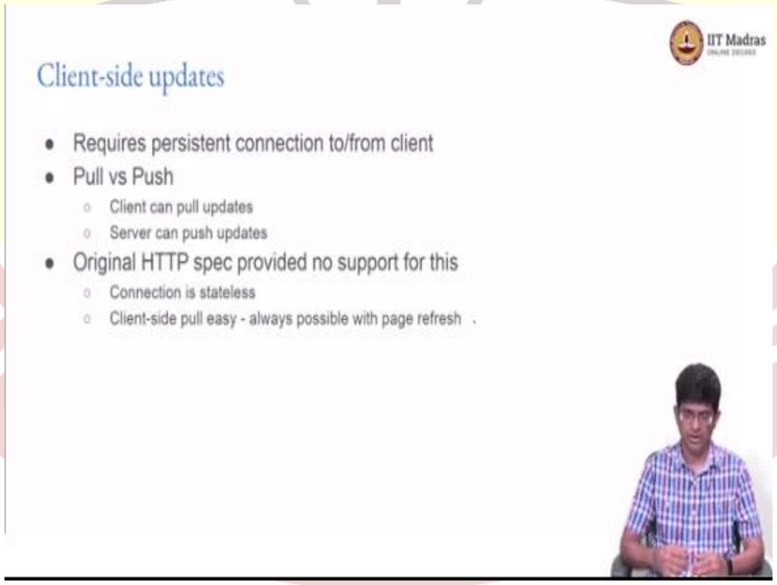


Message targets

- Server to server push
  - Server A requests some messages to be sent
  - Server B calls back webhook after sending all messages
- What about asynchronous push to client?
  - Status updates?
  - Push notifications?

So, server to server push server A request some messages to be sent and server B calls back a web hook after sending those messages that is an example use case. But what happens if I want an asynchronous push to a client status updates, push notifications, and so on.

(Refer Slide Time: 01:47)



Client-side updates

- Requires persistent connection to/from client
- Pull vs Push
  - Client can pull updates
  - Server can push updates
- Original HTTP spec provided no support for this
  - Connection is stateless
  - Client-side pull easy - always possible with page refresh

For getting any kind of client side update, that is to say, if I want, when I say client, what I have in mind, over here is either a web browser that is running on a desktop or a tab, or it could be a mobile phone application. Since we are talking primarily about web applications, we will restrict ourselves more or less to the domain of web based notification systems, although some of the ideas over here apply to any kind of notification system in general.

Now, the point is, any client that connects to a server, a client can connect to a server because it knows the address of the server. It knows some name, some URL googledot com or yahoo dot com or Microsoft dot com. And it knows how to translate that into an IP address, send a request to that server and then get back a response from there.

Now, the server on the other hand, does not know the details of every single client. So, how does the server connect back to a client and say, okay there is a message for you. Why is that so much of a problem? Because the IP address that is assigned to me at any point in time is almost certainly going to change with time, it is not going to be always the same, unless I have registered some kind of a public IP somewhere and made sure that it is constant.

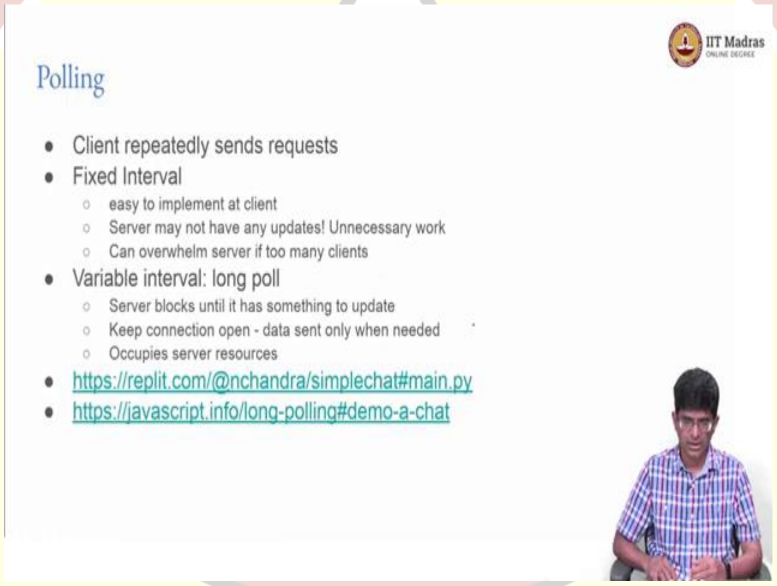
And the moment and not just that, in a lot of cases, especially with internet service providers, the number of globally accessible public IPs is limited, which means that the IP address that you finally get is something of the form 10 dot something something or 190 2.1 68. something, all those fall in what are called private IP address spaces, meaning that they are not accessible directly from the internet, which is why I can have an IP address in the range 10 dot something, and my neighbor can have their own Wi Fi router, which gives them another set of IP addresses and 190 2.1 68.

The person in the third house also has 190 2.1 68. And these do not overlap at all, they might in fact, both have the same address 190 2.1 68 point 1.1. But one cannot reach the other. And the reason is because the IP protocol is meant to drop packets and not route them across the internet. So, what that means is a server cannot directly connect back to a client in general, the only way to do it is if a client connects to the server and then keeps that connection open. It keeps it persistent.

Effectively, if you think about it, what that means is you are providing a mechanism at some level whereby the client can keep checking with the server, whether there is a new message available. In other words, you are polling data. Now, in some sense, that is easiest to do if the client can keep checking, polling data from the server. And you do that often enough, you could give sort of the illusion of being almost a real time notification. But maybe a small amount of delay.

Now the problem with all of this the client side update, why it becomes an issue is that the original HTTP specification does not really consider this usage scenario at all. Connections are meant to be stateless. And client side poll was considered easy to do if at all, you need to get an update from the server, you just request data from the server again. The server should not retain state about which clients are connected to it, what they are looking for, or what information needs to be sent to them that is part of the basic HTTP spec.

(Refer Slide Time: 05:33)



Polling

- Client repeatedly sends requests
- Fixed Interval
  - easy to implement at client
  - Server may not have any updates! Unnecessary work
  - Can overwhelm server if too many clients
- Variable interval: long poll
  - Server blocks until it has something to update
  - Keep connection open - data sent only when needed
  - Occupies server resources
- https://replit.com/@nchandra/simplechat#main.py
- https://javascript.info/long-polling#demo-a-chat

So, how do we handle this? Like I said, the simplest way, in some sense is to use something called polling. And in polling, what we do is that the client repeatedly sends requests, what happens when a client sends a request, the server has to once again look at whatever data it has, and send back information. Now, you could do this in a couple of different ways. One is that you do what is called fixed interval polling.

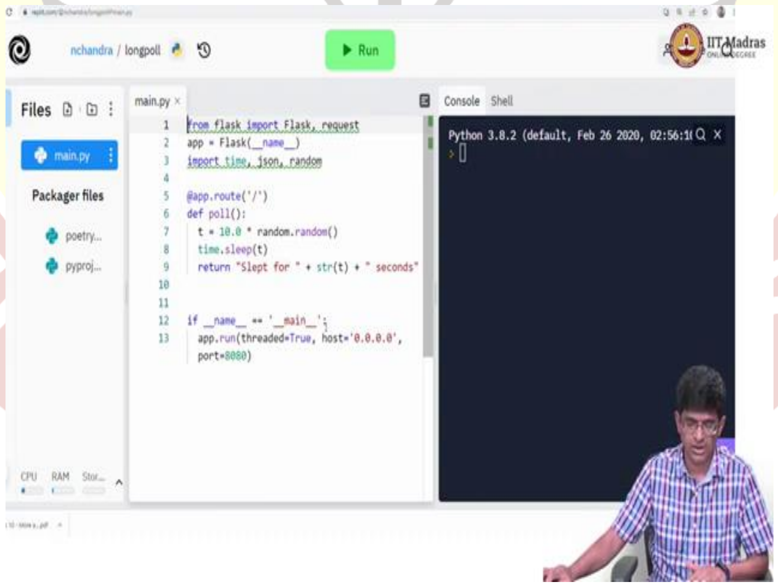
Every second, you basically go and ask the server, any updates? No. Any updates? No. Any updates? No. But it sounds boring but these are machines, they can do them forever, without really getting bored. The alternative is something called a variable interval or a long poll. In a long poll, what you do is a client connects to the server and ask any update, the server does not respond.

What happens when the server does not respond, the client waits, it waits, and it waits and it waits until it finally gets a response. The good part about it is the client is not repeatedly hitting the server asking for updates. The bad part is, once it connects, it is not disconnecting, the client has made a connection and some resources are occupied on the client. Some resources are also occupied on the server.

The server basically has one port blocked because the client in order to establish that communication channel and the client. It also has some memory resources blocked for this connection with the client. All of this is down at the operating system level. So, it may not be very apparent to the person who is developing an application in Python. But you have to keep it in mind when you are actually doing something of this you think that a long pole is very easy to do.

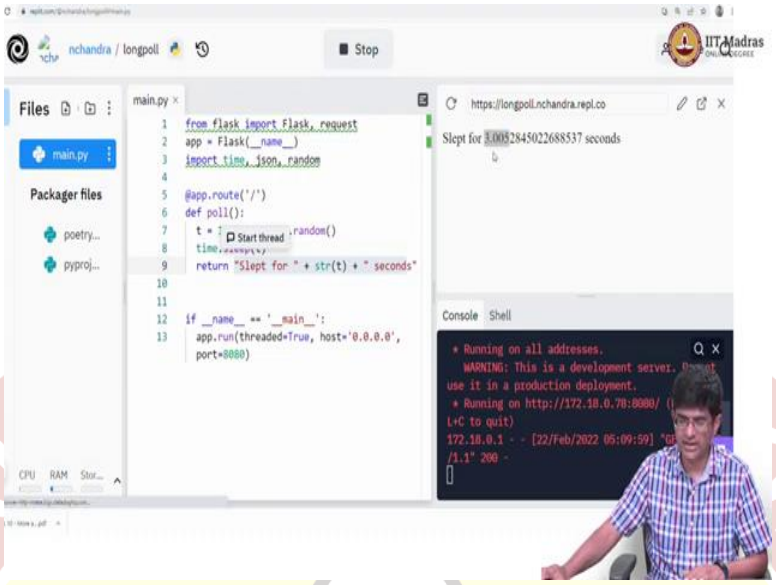
But what is happening in practice is that at the operating system level, it is actually blocking resources. So, a long pole is one sort of very simple way by which you can establish some kind of a mechanism where somebody is just going to wait for a response from you. And you keep the connection open until you have something to tell them.

(Refer Slide Time: 07:41)



```python
from flask import Flask, request
app = Flask(__name__)
import time, json, random

@app.route('/')
def poll():
  t = 10.0 * random.random()
  time.sleep(t)
  return "Slept for " + str(t) + " seconds"


if __name__ == '__main__':
  app.run(threaded=True, host='0.0.0.0',
  port=8080)
```

```
Python 3.8.2 (default, Feb 26 2020, 02:56:1
> 
```

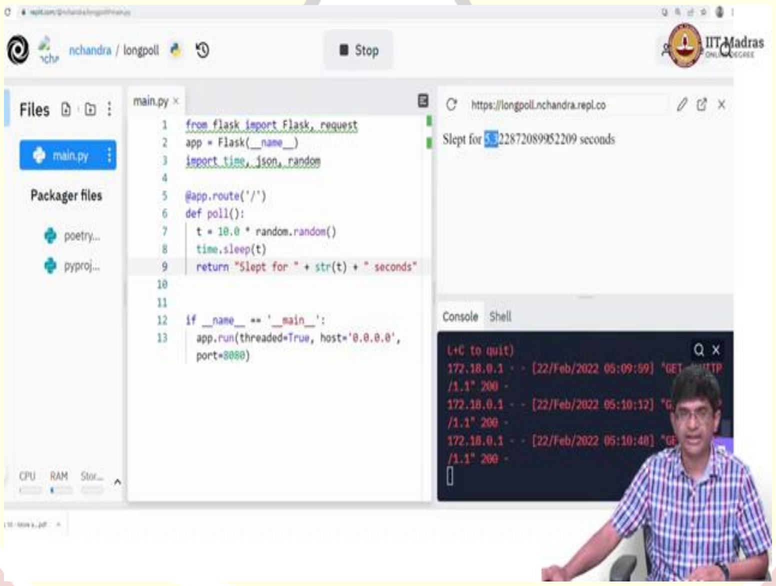


```python
from flask import Flask, request
app = Flask(__name__)
import time, json, random

@app.route('/')
def poll():
    t = 1 Start thread .random()
    time.sleep(t)
    return "Slept for " + str(t) + " seconds"


if __name__ == '__main__':
    app.run(threaded=True, host='0.0.0.0',
    port=8080)
```

https://longpoll.nchandra.repl.co

```
Slept for 3.005284502268537 seconds
```

```
 * Running on all addresses.
   WARNING: This is a development server. Do not
use it in a production deployment.
 * Running on http://172.18.0.78:8080/ (
L+C to quit)
172.18.0.1 - - [22/Feb/2022 05:09:59] "GE
/1.1" 200 -
```



```python
from flask import Flask, request
app = Flask(__name__)
import time, json, random

@app.route('/')
def poll():
    t = 10.0 * random.random()
    time.sleep(t)
    return "Slept for " + str(t) + " seconds"


if __name__ == '__main__':
    app.run(threaded=True, host='0.0.0.0',
    port=8080)
```

https://longpoll.nchandra.repl.co

```
Slept for 5.322872089952209 seconds
```

```
L+C to quit)
172.18.0.1 - - [22/Feb/2022 05:09:59] "GET / HTTP
/1.1" 200 -
172.18.0.1 - - [22/Feb/2022 05:10:12] "G
/1.1" 200 -
172.18.0.1 - - [22/Feb/2022 05:10:48] "GE
/1.1" 200 -
```

So, this is an example of a very trivial flask application that implements some kind of long polling. Effectively, what it does over here is it defines a function called poll. All that, that does is to basically go and sleep for a random amount of time. And only after it has woken up from that, it will come back with a response. What happens if I actually run this? I find that the page long poll, essentially, when it opens, you would notice that it just basically showed for a little bit of time out there and then finally printed out 3.005.

If I refresh the page, it just does not seem to respond for a little while and then after some time, suddenly, this 2.75 seconds comes up. If you look carefully at it, it is not as though I was repeatedly connecting to it once every second to check. It literally just went to sleep for 2.75

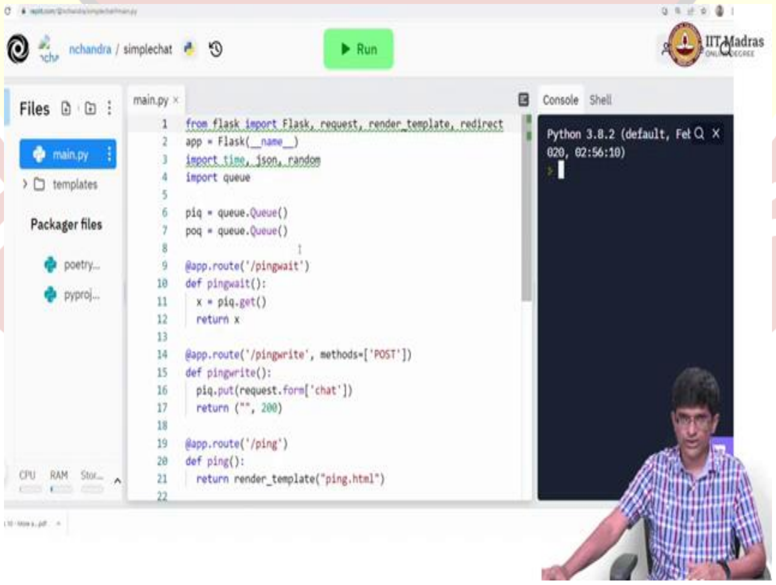
seconds. This large number of decimals, you should probably ignore it. It is just an artifact of how Python prints floating point numbers.

It is not accurate to that level definitely. But around 2.75 seconds, you can say yes, it actually waited for that long before giving me a response. What happens very low rate again, it just waits and waits and waits. I do not know how long it is going to take this time after all, it is random. But after 5.3 seconds, I get a response. Even though it was 5 seconds, I only had to make one request, whether it was 2 seconds or 5 seconds or 10 seconds, I had to make one request. The server basically kept me blocked over there until it could finally send me back information.

Now, what if I combined this with something which is a piece of JavaScript on my client end that literally just tried to reload the page or re-fetch from here, every time after it got a request, got an update? Effectively, that would be some way by which I can have a continuous poll of the system. The server only needs to send me back a response anytime it has an update. As soon as it sends me an update, I sent it the next request telling me okay fine, you know I got this update now send me the next request and wait.

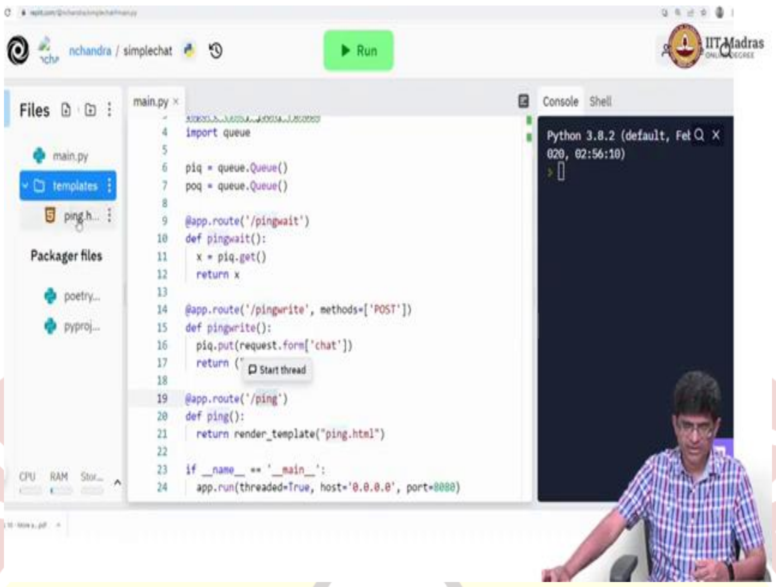
We can use this in order to build up something which is sort of a trivial form of a chat application. We will look at that as the next demo.

(Refer Slide Time: 10:00)



```python
from flask import Flask, request, render_template, redirect
app = Flask(__name__)
import time, json, random
import queue

piq = queue.Queue()
poq = queue.Queue()

@app.route('/pingwait')
def pingwait():
  x = piq.get()
  return x

@app.route('/pingwrite', methods=['POST'])
def pingwrite():
  piq.put(request.form['chat'])
  return ("", 200)

@app.route('/ping')
def ping():
  return render_template("ping.html")
```

```
Python 3.8.2 (default, Fet
020, 02:56:10)
> 
```



```python
import queue

piq = queue.Queue()
poq = queue.Queue()

@app.route('/pingwait')
def pingwait():
  x = piq.get()
  return x

@app.route('/pingwrite', methods=['POST'])
def pingwrite():
  piq.put(request.form['chat'])
  return ('

@app.route('/ping')
def ping():
  return render_template("ping.html")

if __name__ == '__main__':
  app.run(threaded=True, host='0.0.0.0', port=8080)
```

```
Python 3.8.2 (default, Fel
020, 02:56:10)
> 
```

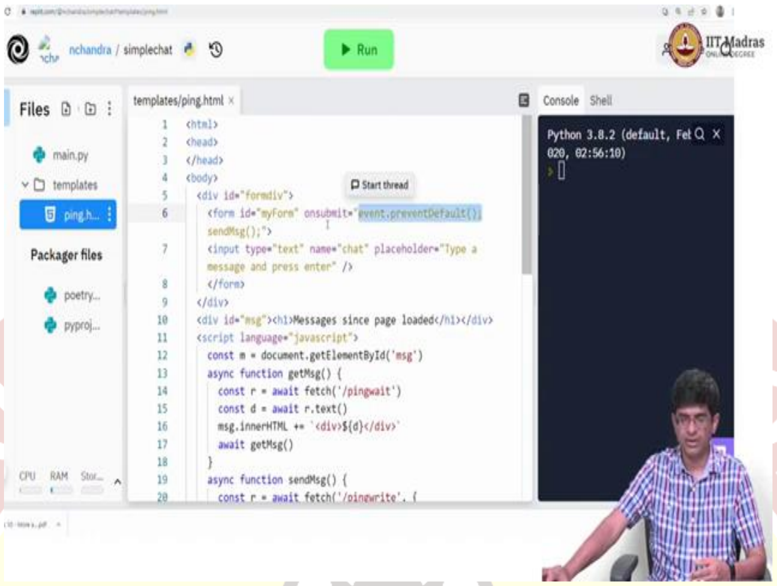
So, this chat application that I am showing over here is a little bit more of a complicated, a slightly more complicated example that I have. It, of course, uses flask but in addition to that, it also uses a little bit of JavaScript which I will show you in a moment. As you can see, there are a few different routes are here, I am defining something called ping wait. Which, when you look at it, it is doing something strange. It is doing something called a get request out here.

What exactly is this GET request? If you look at it, I have defined something called, I am using something called import queue and this is part of the Python standard library, there is something called a queue. And what happens with a queue is that I can define a queue and when I do a get on that queue, it will block and wait until there is some data on the queue. A queue can be thought of as a list. Initially, the list is empty it does not have any information in it.

But what this get does is it tries to get some data out of the list, initially, it is empty, so it can not get anything, which means that it just blocks and waits over here. But there is another function that basically says, any time that I write to this URL, ping right take some information out of a form, we will get to what this form is in a moment, and put it onto this queue, the piq. And in this case, basically just return nothing do not return any information out here, just accept the data in other words, just return an HTTP 200 code so that everybody is happy.

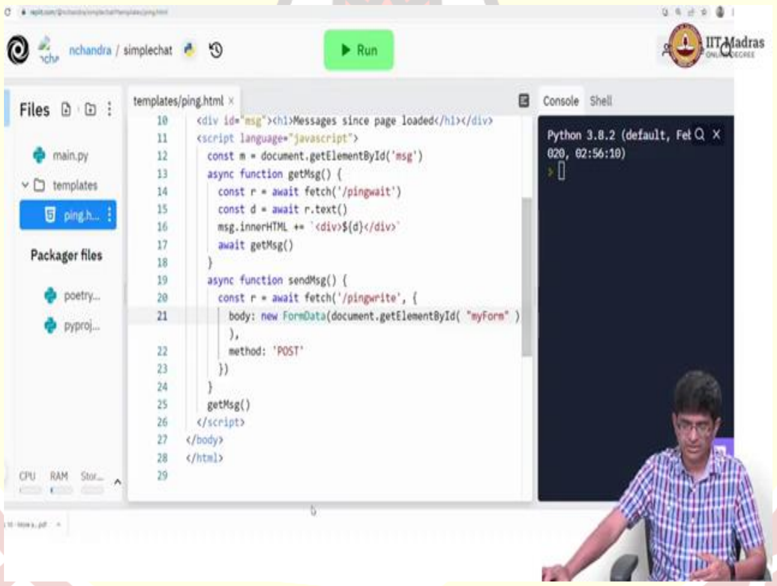
And finally, of course, I have the HTML page out here, that gets rendered when I actually go to the URL slash ping. So, let us look at what the HTML page looks like.

(Refer Slide Time: 12:00)



```
<html>
<head>
</head>
<body>
  <div id="formdiv">
    <form id="myForm" onsubmit="event.preventDefault();
    sendMsg();">
    <input type="text" name="chat" placeholder="Type a
    message and press enter" />
    </form>
  </div>
  <div id="msg"><h1>Messages since page loaded</h1></div>
  <script language="javascript">
    const m = document.getElementById('msg')
    async function getMsg() {
      const r = await fetch('/pingwait')
      const d = await r.text()
      msg.innerHTML += `<div>${d}</div>`
      await getMsg()
    }
    async function sendMsg() {
      const r = await fetch('/pingwrite', {
```

```
Python 3.8.2 (default, Fe
020, 02:56:10)
> 
```



```
  <div id="msg"><h1>Messages since page loaded</h1></div>
  <script language="javascript">
    const m = document.getElementById('msg')
    async function getMsg() {
      const r = await fetch('/pingwait')
      const d = await r.text()
      msg.innerHTML += `<div>${d}</div>`
      await getMsg()
    }
    async function sendMsg() {
      const r = await fetch('/pingwrite', {
        body: new FormData(document.getElementById( "myForm" )
        ),
        method: 'POST'
      })
    }
    getMsg()
  </script>
</body>
</html>
```

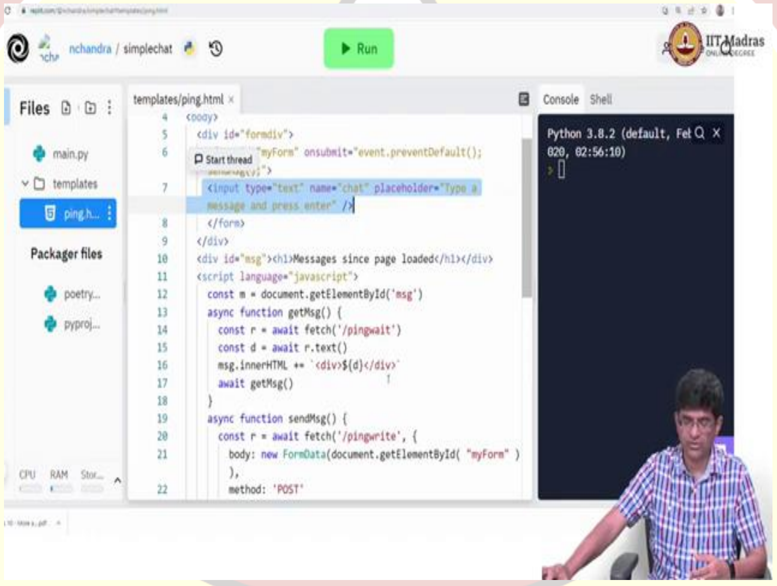
```
Python 3.8.2 (default, Fe
020, 02:56:10)
> 
```

If I look at the HTML page out here, I find that what I am doing is I am defining a formdiv. And, in particular, defining a form then, if you look at it, what I am saying is on submit, that is to say, when I enter data in the form, or click the submit button, this form does not have a submit button. But so what I need to do is type something into the input field and press enter. What it will do is it will prevent the default. What does that mean?

The default mode of operation of a form is to actually make an HTTP post request to the URL from which it was called. So, if I had not specified this event dot prevent default, what would have happened is when I press enter after typing in something into the chat, or into the text box, it would have called my slash ping URL, once again, with a post request and the point is, I do not actually know what to do with a slash with a post request to slash ping.

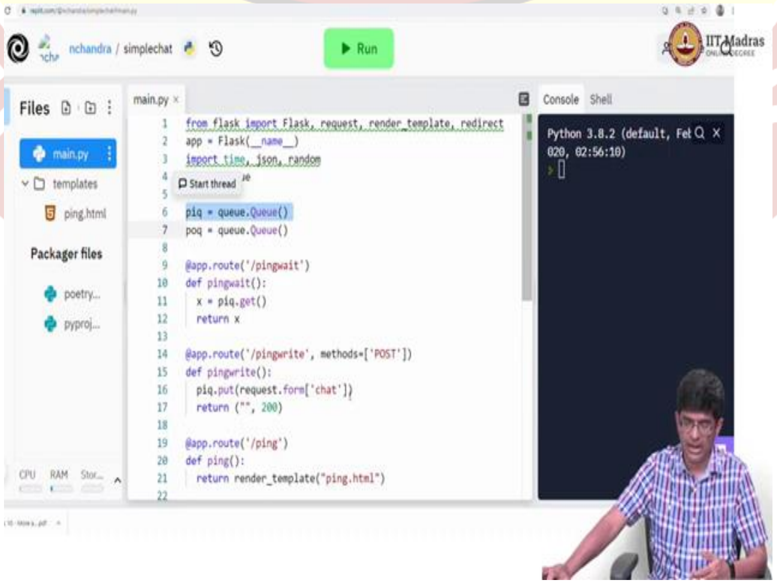
So, I am telling the JavaScript to prevent that from happening in the first place, instead, to call this function send message. What does send message do? It basically, it is an async function. Why is it an async function? Because I need to call fetch which is an async operation. And all that I do is I basically say await fetch slash ping right. And what do I write in it? I am going to take the form data whatever was there in this form and post it as the body of my fetch request to slash ping right.

(Refer Slide Time: 13:39)



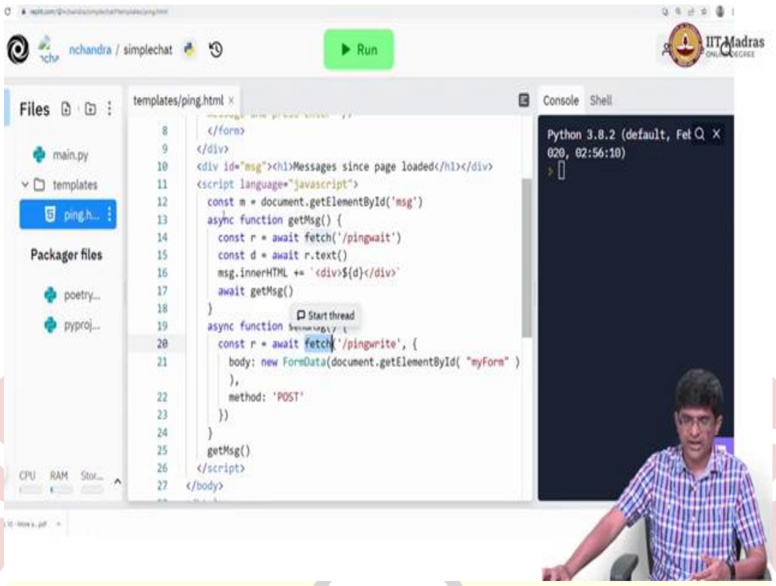
```
<body>
  <div id="formdiv">
    "myForm" onsubmit="event.preventDefault();
      sendMsg();">
      <input type="text" name="chat" placeholder="Type a
      message and press enter" />
    </form>
  </div>
  <div id="msg"><h1>Messages since page loaded</h1></div>
  <script language="javascript">
    const m = document.getElementById('msg')
    async function getMsg() {
      const r = await fetch('/pingwait')
      const d = await r.text()
      msg.innerHTML += `<div>${d}</div>`
      await getMsg()
    }
    async function sendMsg() {
      const r = await fetch('/pingwrite', {
        body: new FormData(document.getElementById( "myForm" )
        ),
        method: 'POST'
```

```
Python 3.8.2 (default, Feb
020, 02:56:10)
> 
```



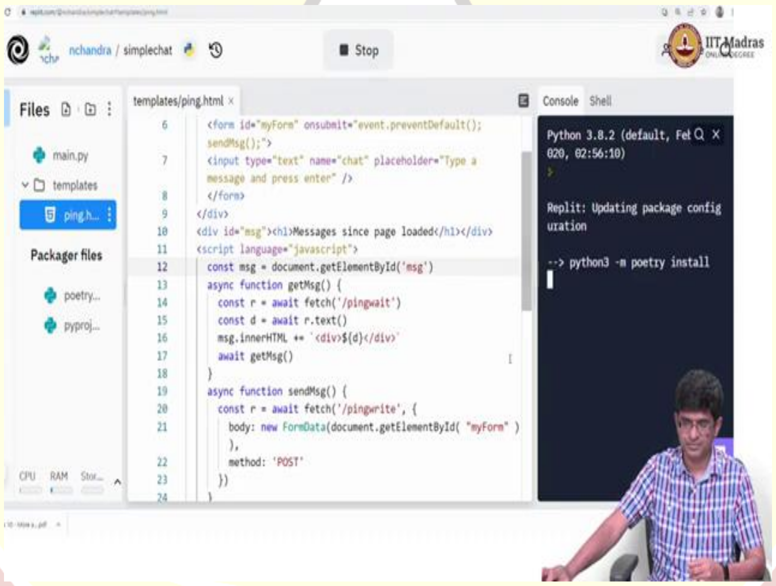
```
from flask import Flask, request, render_template, redirect
app = Flask(__name__)
import time, json, random
    ue

piq = queue.Queue()
poq = queue.Queue()

@app.route('/pingwait')
def pingwait():
  x = piq.get()
  return x

@app.route('/pingwrite', methods=['POST'])
def pingwrite():
  piq.put(request.form['chat'])
  return ("", 200)

@app.route('/ping')
def ping():
  return render_template("ping.html")
```

```
Python 3.8.2 (default, Feb
020, 02:56:10)
> 
```



```
    </form>
  </div>
  <div id="msg"><h1>Messages since page loaded</h1></div>
  <script language="javascript">
    const m = document.getElementById('msg')
    async function getMsg() {
      const r = await fetch('/pingwait')
      const d = await r.text()
      msg.innerHTML += `<div>${d}</div>`
      await getMsg()
    }
    async function sendMsg() {
      const r = await fetch('/pingwrite', {
        body: new FormData(document.getElementById( "myForm" )
        ),
        method: 'POST'
      })
    }
    getMsg()
  </script>
</body>
```

```
Python 3.8.2 (default, Fe
020, 02:56:10)
> 
```



```
    <form id="myForm" onsubmit="event.preventDefault();
    sendMsg();">
    <input type="text" name="chat" placeholder="Type a
    message and press enter" />
    </form>
  </div>
  <div id="msg"><h1>Messages since page loaded</h1></div>
  <script language="javascript">
    const msg = document.getElementById('msg')
    async function getMsg() {
      const r = await fetch('/pingwait')
      const d = await r.text()
      msg.innerHTML += `<div>${d}</div>`
      await getMsg()
    }
    async function sendMsg() {
      const r = await fetch('/pingwrite', {
        body: new FormData(document.getElementById( "myForm" )
        ),
        method: 'POST'
      })
    }
```

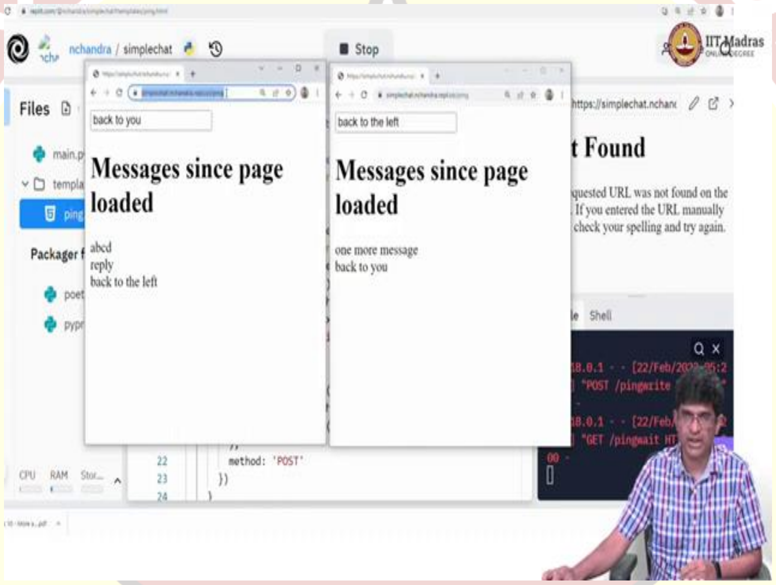
```
Python 3.8.2 (default, Fe
020, 02:56:10)
>

Replit: Updating package config
uration

--> python3 -m poetry install
```

What will happen as a result in ping right, I will look for the entry chat, which after all, was the name that I had given for this input field out here. Take whatever was the information that I got. And in ping right I will put it onto this queue, piq and just return. So, in other words, you will notice that I am awaiting this fetch but I am not really awaiting the response of the fetch. Normally, I should also await the whatever r dot JSON or r dot text or something, I am not really I do not care.

I am in this case, I am not going to make any use of that. I am just awaiting the fetch itself to come back and that is all that I really care about. Now, I have one more function in my JavaScript, which basically says, get message what does get message do? It awaits a fetch to ping wait, what does ping wait do? It tries to look for data present on the queue? It does nothing. It waits which means that I will wait on ping wait until I get something as soon as I get it. I convert that into text. I add it to the MSG. inner HTML where MSG itself is defined as this I think it should have been m. inner HTML.

I think we can just do that MSG going to this. And I have this entire thing set up to run over here, MSG is equal to document dot get element by Id MSG and message dot inner HTML plus equal to this div what happened when I run this.

(Refer Slide Time: 15:38)



back to you

Messages since page loaded

abcd
reply
back to the left

back to the left

Messages since page loaded

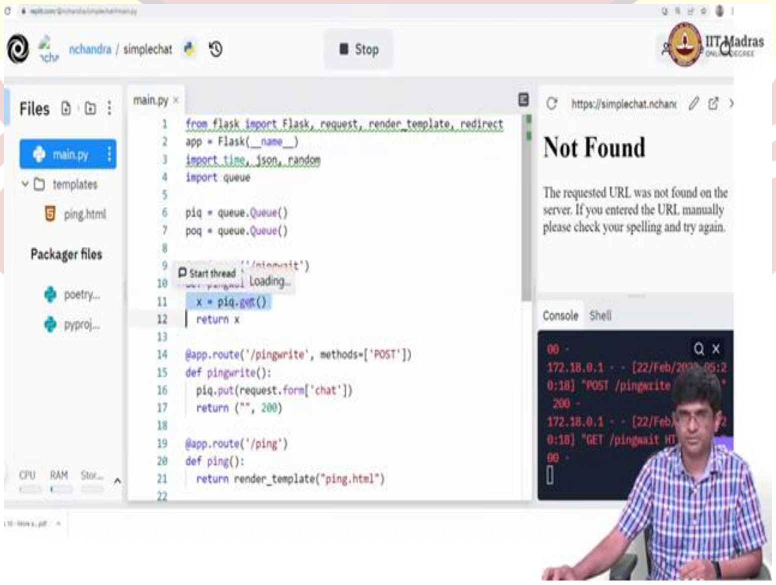
one more message
back to you

```
        method: 'POST'
    })
}
```

t Found

quested URL was not found on the . If you entered the URL manually check your spelling and try again.

```
18.0.1 - - [22/Feb/20__ 05:2
] "POST /pingwrite
18.0.1 - - [22/Feb/
] "GET /pingwait HT
00 -
```

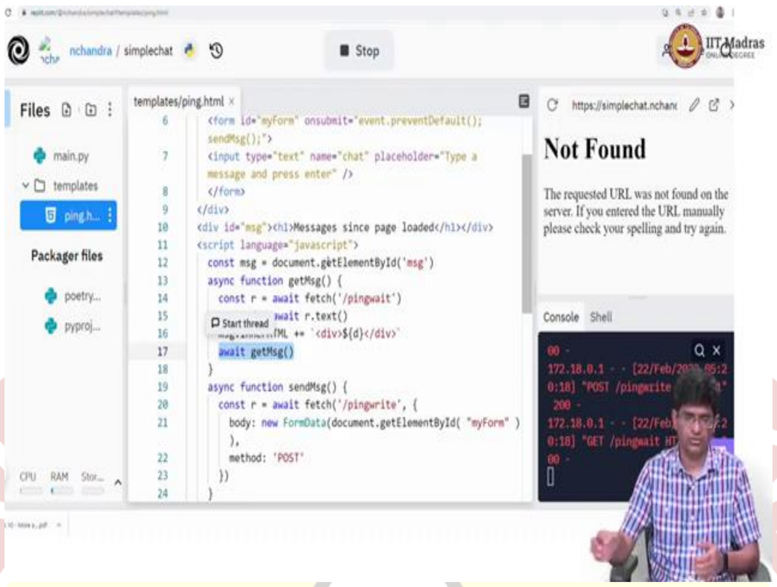


```
from flask import Flask, request, render_template, redirect
app = Flask(__name__)
import time, json, random
import queue

piq = queue.Queue()
poq = queue.Queue()

            ('/pingwait')
def pingwai
  x = piq.get()
  return x

@app.route('/pingwrite', methods=['POST'])
def pingwrite():
  piq.put(request.form['chat'])
  return ("", 200)

@app.route('/ping')
def ping():
  return render_template("ping.html")
```

Not Found

The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

```
00 -
172.18.0.1 - - [22/Feb/20__ 05:2
0:18] "POST /pingwrite
 200 -
172.18.0.1 - - [22/Feb/
0:18] "GET /pingwait HT
00 -
```



```
    <form id="myForm" onsubmit="event.preventDefault();
    sendMsg();">
    <input type="text" name="chat" placeholder="Type a
    message and press enter" />
    </form>
  </div>
  <div id="msg"><h1>Messages since page loaded</h1></div>
  <script language="javascript">
    const msg = document.getElementById('msg')
    async function getMsg() {
      const r = await fetch('/pingwait')
      [illegible] await r.text()
      [illegible]TML += `<div>${d}</div>`
      await getMsg()
    }
    async function sendMsg() {
      const r = await fetch('/pingwrite', {
        body: new FormData(document.getElementById( "myForm" )
        ),
        method: 'POST'
      })
    }
```

Not Found

The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

```
00 -
172.18.0.1 - - [22/Feb/20[illegible] 05:2
0:18] "POST /pingwrite [illegible]"
 200 -
172.18.0.1 - - [22/Feb[illegible]
0:18] "GET /pingwait H[illegible]
00 -
```

So, what I have done now is I ran the replit corresponding to the simple chat application, and I have opened two browser windows. And what I have over here is both of them are just showing a messaging message since page loaded. Let me try typing in a message out here. And I find that it actually ends up coming on the same screen that I had. Why is that? If you think about it, since the page on the left was the first one to have been opened, it was the first one to make a call to the slash ping wait.

Which means that the first message that was posted over there actually ended up coming on that same screen itself. So, I am going to just type in one more message and this time, I find that the message actually goes to the other screen. Why is that? Because the second ping wait a second call to ping wait, what had actually been made by the right hand side browser window. So, now I can go to the right hand side browser window, and type in a reply, which I find goes back to the left hand side of the screen.

And now I can keep sending messages back and forth and I find that messages alternate nicely over here. So, what has happened, effectively, I have these two windows, both of which have connected to the same URL slash ping. And because of the fact that in my Replit, I had a queue that had been set up. And the ping wait would just basically wait for information on that queue. Remove it from the queue, as soon as it finds something, and add it and send it back on a URL.

And my JavaScript in turn, essentially, all that it would do is it would wait for some data on ping, wait, it would then take the text added to the message inner HTML and now here's the crucial part. It once again goes back and awaits get message. So, it is sort of you can think of it as a recursive endless waiting call. As soon as get messages done, it goes back and calls itself one more time. And as a result of this looping back to the beginning, what is happening is it is effectively sitting there in an infinite loop waiting for data.

Anytime that something is present on the queue, it will de-queue it get the information and proceed. What does it do with the information? It just basically goes, adds it to its browser, whichever one that particular JavaScript corresponded to and once again, go back and start to look. So, as you can see over here, this is a very trivial form of, you can think of it as a kind of a chat window, I mean I have two windows that are open.

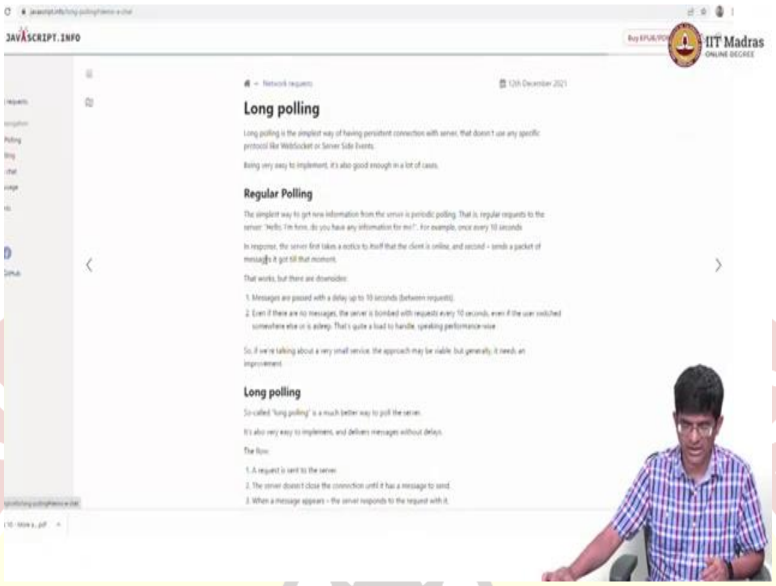
I opened the both on the same desktop, but it could have been one on my phone and one on somebody else's phone. And I could have used this in order to send messages back and forth. Now, obviously, this is terribly buggy, for one very obvious reason, which was actually there in the very first message that we tried sending. The first message that is sent actually comes back to the person sending it. Why did that happen? Because all that I am doing is I am blindly waiting for, no, a queue, I am just taking whatever I get from the queue and displaying it.

Now, obviously, this can be fixed. One way of doing it might have been that along with this ping wait, I could also have had another pong wait. So, a ping pong kind of setup where I use another, another queue. So, whenever I access the ping URL, it adds messages onto the PO queue. And the pong URL will add messages to the PI queue. That can sort of be used in order to ensure that message is actually go to the opposite person.

How do I make sure that multiple people do not connect to ping? And they can, I can prevent that from happening. I could in turn have in my code, something which says that if ping has already been connected to I sent a global flag and say, hey, this is already busy, you cannot do this. But that makes it very restrictive, I can only communicate between two people who I need to tell them the URLs in advance.

So, obviously, this is a very sort of overly simplistic form of setting things up. The interesting thing is, is actually derived from an example out here.

(Refer Slide Time: 21:13)



Network requests

12th December 2021

Long polling

Long polling is the simplest way of having persistent connection with server, that doesn't use any specific protocol like WebSocket or Server Side Events.

Being very easy to implement, it's also good enough in a lot of cases.

Regular Polling

The simplest way to get new information from the server is periodic polling. That is, regular requests to the server: "Hello, I'm here, do you have any information for me?". For example, once every 10 seconds.

In response, the server first takes a notice to itself that the client is online, and second – sends a packet of messages it got till that moment.

That works, but there are downsides:

1. Messages are passed with a delay up to 10 seconds (between requests).
2. Even if there are no messages, the server is bombed with requests every 10 seconds, even if the user switched somewhere else or is asleep. That's quite a load to handle, speaking performance-wise.

So, if we're talking about a very small service, the approach may be viable, but generally, it needs an improvement.

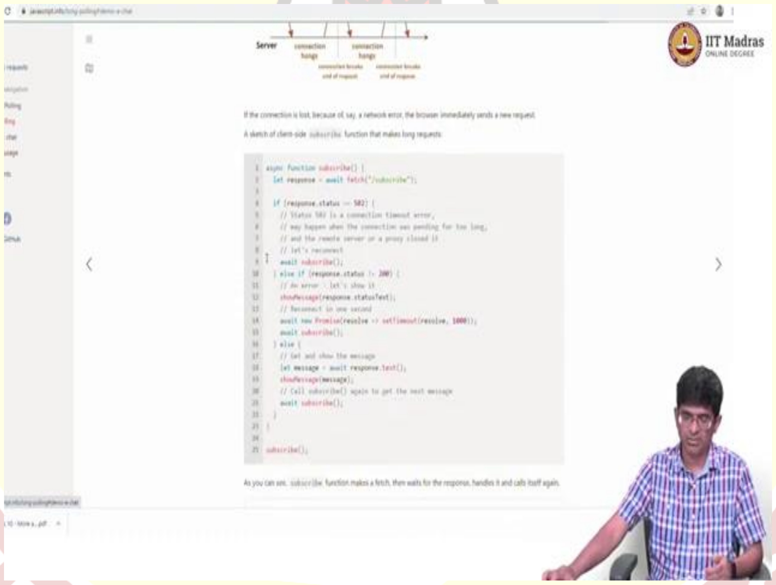
Long polling

So-called "long polling" is a much better way to poll the server.

It's also very easy to implement, and delivers messages without delays.

The flow:

1. A request is sent to the server.
2. The server doesn't close the connection until it has a message to send.
3. When a message appears – the server responds to the request with it.



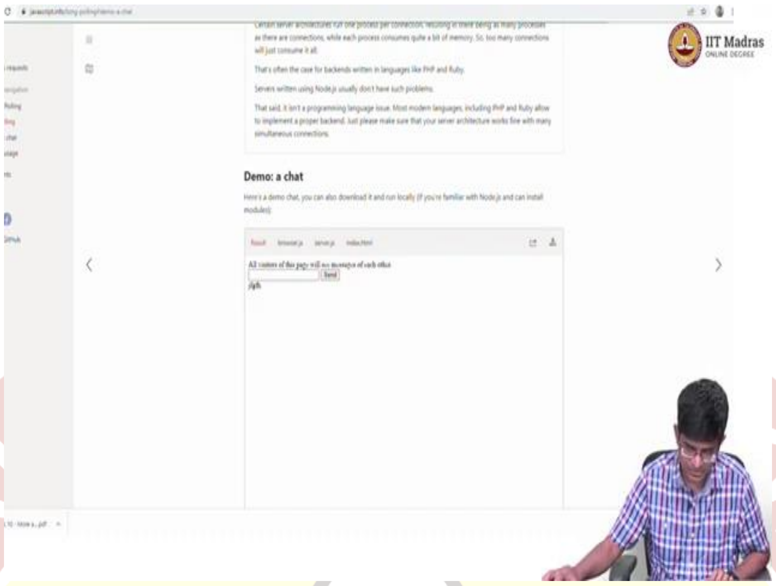
Server
connection hangs
connection hangs
connection breaks end of request
connection breaks end of response

If the connection is lost, because of, say, a network error, the browser immediately sends a new request.

A sketch of client-side `subscribe` function that makes long requests:

```
async function subscribe() {
  let response = await fetch("/subscribe");

  if (response.status == 502) {
    // Status 502 is a connection timeout error,
    // may happen when the connection was pending for too long,
    // and the remote server or a proxy closed it
    // let's reconnect
    await subscribe();
  } else if (response.status != 200) {
    // An error - let's show it
    showMessage(response.statusText);
    // Reconnect in one second
    await new Promise(resolve => setTimeout(resolve, 1000));
    await subscribe();
  } else {
    // Get and show the message
    let message = await response.text();
    showMessage(message);
    // Call subscribe() again to get the next message
    await subscribe();
  }
}

subscribe();
```

As you can see, `subscribe` function makes a fetch, then waits for the response, handles it and calls itself again.


javascript.info/long-polling#demo-a-chat

Certain server architectures run one process per connection, resulting in there being as many processes as there are connections, while each process consumes quite a bit of memory. So, too many connections will just consume it all.

That's often the case for backends written in languages like PHP and Ruby.

Servers written using Node.js usually don't have such problems.

That said, it isn't a programming language issue. Most modern languages, including PHP and Ruby allow to implement a proper backend. Just please make sure that your server architecture works fine with many simultaneous connections.

Demo: a chat

Here's a demo chat, you can also download it and run locally (if you're familiar with Node.js and can install modules):

Result browser.js server.js index.html

All visitors of this page will see messages of each other

Send

jfgfh

Which is from JavaScript dot info, and which talks about long polling and how to set it up. And in long polling, they have given a completely JavaScript based example, where you can actually have a similar kind of a demo. Where you could open this and now what happens is, all the visitors of the page will actually see the messages are being sent to this chat.

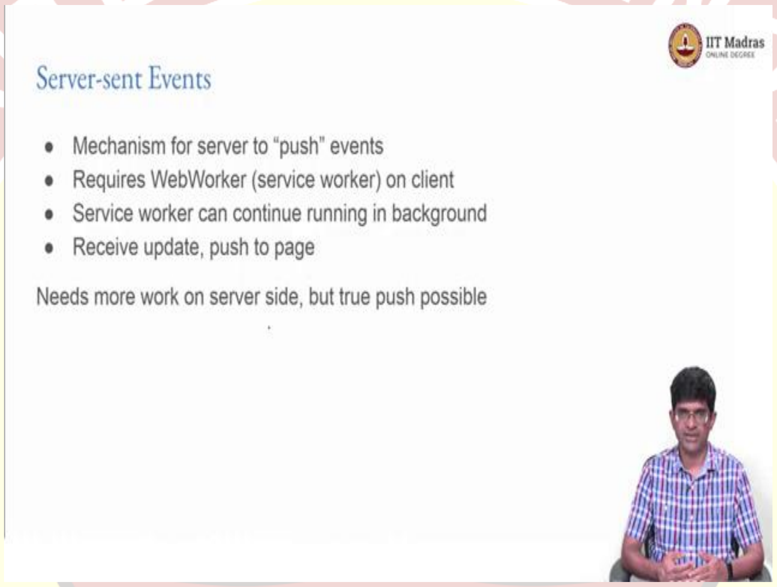
Once again, in principle, when each one of these opens it up separately, you will find that you could actually have a chat application, each person connecting to this gets a sort of a unique subscriber ID. And when you look at the JavaScript, you will see that it maintains a list of who are all the people connected and sends information to all of them.

That has its own problems, what happens if somebody closes a browser window? I should be able to sort of take that off the list and say, okay I no longer need to send messages out here. So, the having said that the moment I close the browser window, it also means that I will no longer repeat the long pole, I will no longer come back and try to connect. So, at most, I might send a message once and then that is it, I will not try to connect again after that.

As you can see, this is a very trivial way of implementing a chat functionality. It has all the requirements that you have, effectively, what happens is a real time notification is being sent. How is it being done? The browser, the client is essentially establishing a connection to the server, which is being kept open until the server actually has information to send back to it. The biggest problem with this is the fact that it sort of ties up resources on the server.

Having said that, the resources have to be tied up at some level, whichever way you look at it. Anytime a server, even if it just needs to be able to push notifications back to a client, it needs to have information about which client, where they are and how to send that information. So, if the connection, all that it does is it requires a similar amount of information to what was required by the server, if it needed to push information out there, then this might actually be worth it. And this is a reasonable way of implementing push notification, in many simple instances.

(Refer Slide Time: 23:42)



Server-sent Events

- Mechanism for server to "push" events
- Requires WebWorker (service worker) on client
- Service worker can continue running in background
- Receive update, push to page

Needs more work on server side, but true push possible

Having said all of that, about long polling, there are technically superior methods that are available for implementing it, especially in today's browsers. And one of the mechanisms that is there for this is what is called Server Sent Events. A server sent event is something that explicitly allows a server to send back a notification. Now, one of the issues with the long polling is that it works, as long as the page with the long pole or with the, that is being considered is in the foreground, on the front of the screen.

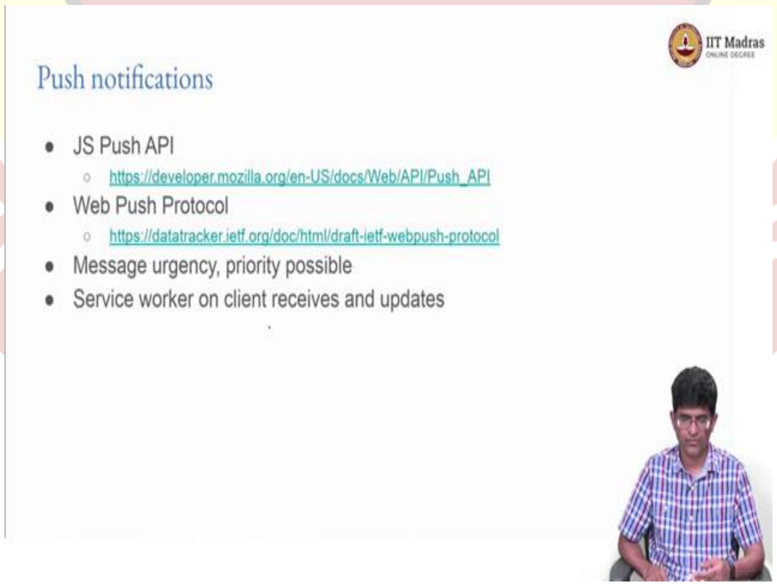
The moment I put it into the background, or in the case of a mobile phone, I switch to another application, in general, these kinds of pages are set so that they are not continuing to perform any activity in the background. So, they sort of go to sleep. Which means that, if I navigate away from this page, that is it, I am not going to get any further notifications. What do I do in such a scenario?

I need some mechanism by which there is someone on my side who is guaranteed to be listening and waiting for messages from a server. And that mechanism is usually some kind of a service worker that can be registered in JavaScript. So, Server Sent Events allow you to have a service worker that can go back into the background, continue running even though you have navigated away from the page or have gone to some other application altogether on a phone.

And the moment that it receives something over here, it can actually push an update onto the page that you have. And if your operating system allows it, it might even be able to push a notification onto your main screen. That all depends on how the interaction with the operating system is and what kind of functionality the service worker has been given. The important point is that the service worker by itself can run in the background, and continue to do so even when you are not on that particular page.

This requires a little bit more work on the server side to set up the server side events. You need to have some kind of a stream which needs to be registered and so on. And on the client side where you actually need to have a service worker. But both of these sound more complicated than they are I mean, they are, they are fairly simple starter codes that can get you through to implementing something of this sort.

(Refer Slide Time: 26:01)

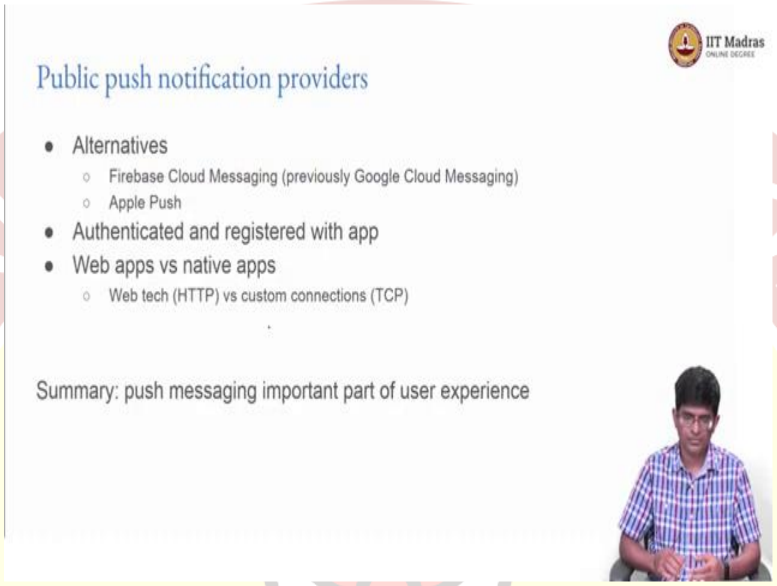


Push notifications

- JS Push API
  - https://developer.mozilla.org/en-US/docs/Web/API/Push_API
- Web Push Protocol
  - https://datatracker.ietf.org/doc/html/draft-ietf-webpush-protocol
- Message urgency, priority possible
- Service worker on client receives and updates

On top of this, of course, you could have full blown what are called push notifications. So, JavaScript by itself has something called a push API. And there is the Web Push Protocol, which allows you to have additional information such as the urgency of a message, the priority associated with the message, all of that can be linked. Ultimately what happens is there has to be a service worker on the client that receives and updates the application.

(Refer Slide Time: 26:30)



Public push notification providers

- Alternatives
  - Firebase Cloud Messaging (previously Google Cloud Messaging)
  - Apple Push
- Authenticated and registered with app
- Web apps vs native apps
  - Web tech (HTTP) vs custom connections (TCP)

Summary: push messaging important part of user experience

Now, finally, what we have talked about is web based mechanisms. There are alternatives that can use public push notification providers. You might have heard the terms Firebase Cloud Messaging FCM, previously, Google Cloud Messaging. Similarly, Apple provides their own push services, push notification services. What happens is that, these are more tightly coupled either with a web application or even with the operating system.

In the case of Apple, that allows you to directly push notifications to applications. There is also a difference in the way that some of the communication works. Some of these things, of course, work over the native HTTP over regular HTTP. But some use some kind of customized connections, especially in the case of native applications, you might be willing to use customized TCP level connections in order to accept these kinds of messages.

So, to summarize all of this, the ability to push messages is a very important part of the user experience for application development. Anytime you want to be able to update a user asynchronously, ideally in real time, when there is something that they need to go and see. And there are multiple mechanisms for doing this polling, long polling, Server Sent Events, push notifications and so on, which require varying degrees of sort of cooperation or support from the operating system or the client. The bottom line is that while some of them might be complicated, they provide a significant enhancement of what the user experience comes across us.