Lab11

Some links for the lab in week 11

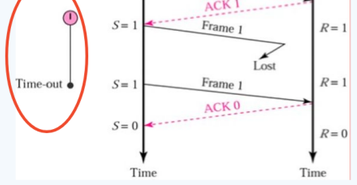
1. Python Web client access. See the LinkedIn learning module:
   Web access with urllib (https://www.linkedin.com/learning/python-xml-json-and-the-web/introducing-urllib?u=2104756)
   which creates Python web clients, that get their information from **http://httpbin.com** (http://httpbin.com) which is a great debugging aid as it delivers properly formatted files for you to test you code on. See the learning module for how this is done.

2. How Stop-and-wait ARQ works: how to use sequence number and ACK/NAK

Stop-and-wait automatic repeat-request (ARQ)

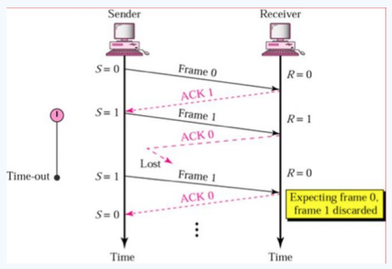
- A stop-and-wait ARQ sender sends **one frame at a time**
- After sending each frame, the sender doesn't send any further frames until it receives an acknowledgement signal (ACK).
  - If there were errors in the received frame, the receiver send a negative acknowledgment signal (NAK). A negative acknowledgement contains no sequence number, the sender just retransmits the last frame sent.
  - After receiving a valid frame, the receiver sends an ACK.
- If the ACK or NAK does not reach the sender before a certain time, known as the **timeout**, the sender sends the same frame again.
- The timeout countdown is reset after each frame transmission.

Stop-and-wait ARQ — lost frame



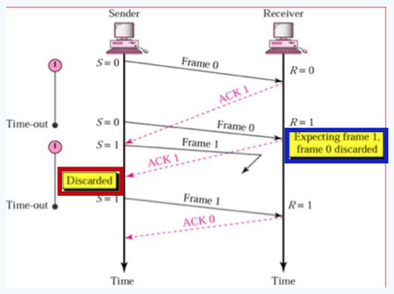
In case a frame never got to receiver, sender has a timer: each time a frame is sent, timer is set.

Stop-and-wait ARQ — lost ACK



If no ACK or NAK is received during timeout period, it re-sends the frame.

Stop-and-wait ARQ — delayed ACK



Suppose timeout and sender retransmits a frame but receiver actually received the previous transmission → receiver has duplicated copies.

To avoid receiving and accepting two copies of same frame, frames and ACKs are alternatively labelled 0 or 1: ACK0 for frame 1 (valid and received) , ACK1 for frame 0.

3. Setting up a Python web server on your PC
   Python: Let's Create a Simple HTTP Server (Tutorial) (https://www.afternerd.com/blog/python-http-server/)

4. Using a **Simple HTTP Handler** (https://docs.python.org/2/library/simplehttpserver.html)

5. Python Documentation on http.server, which lists the variables that you can use in the do_GET() code below.
   https://docs.python.org/3/library/http.server.html#http.server.BaseHTTPRequestHandler
   (https://docs.python.org/3/library/http.server.html#http.server.BaseHTTPRequestHandler)
6. Another copy of **microbitpc.py** (https://rmit.instructure.com/courses/67319/files/14841142/download?wrap=1)

Setting up a web server

Consider the code below.

```
from http.server import HTTPServer, BaseHTTPRequestHandler

class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b'Hello World')

httpd = HTTPServer(('localhost', 8000), SimpleHTTPRequestHandler)

print("Starting server")
httpd.serve_forever()
```

This is the minimal web server. It works like this:

1. We import the HTTP server code which sets up the server and port, and BaseHTTPRequestHandler which is normally called to handle a basic request once the server has been set up and as received a request.
2. We then create a substitute object class which is called before the BaseHTTPRequestHandler, and create a function within it called do_GET() which is the function called when a request arrives. In the case below, this function first sets the return code to 200 then signifies that there are no other changes needed to be made the response headers, and so outputs them. It then outputs the "Hello World" string
3. We then create a particular instance of this new class called httpd by linking i it to an existing class HTTPServer, but substitute the handler to be our own handler.
4. Finally , after confirming that we are set to go ...
5. We tell the server to serve forever.

If you run this code, you then go to a browser and enter **http://localhost:8000** (http://localhost:8000) . Note that it is **NOT https**, as this server is not secure.

You will see "Hello World" on your browser, and a log entry where the code is running.

As an experiment, we could insert a print statement as the first line of code in the do_GET() as

```
        print(self.headers)
```

and we will see the following appear on the console

Starting server
Host: localhost:8000
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:81.0) Gecko/20100101 Firefox/81.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0

These are the request header that you sent from your browser (Firefox in my case). Now to the assignment...

Resetting your micro:bit

The code below will reset your microbit. Note that there is no testing of the URL path, so it is unconditional at the moment. You execute this on your python REPL, and then type **http://localhost:8000 (http://localhost:8000)** on you browser.

```
from http.server import HTTPServer, BaseHTTPRequestHandler
from microbitpc import openMB, resetMB

class SimpleHTTPRequestHandler(BaseHTTPRequestHandler):

    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b'Resetting your Microbit')
        com = openMB()
        resetMB(com)

httpd = HTTPServer(('localhost', 8000), SimpleHTTPRequestHandler)

print("Starting server")
httpd.serve_forever()
```

You can see how little needed to be changed to make this work. But now it is ONLY resetting the MB. The code is too simple.

We need to add some other codes, and also implement a "default action" if the path is not special.

Consider the code below. There are two key changes. First of all BaseHTTLRequestHandler is too simple and has no default behavior. So let's use SimpleHTTPRequestHandler instead. This one will satisfy the path part in the URL, and return any files it finds that match. This is relative to the directory that the server started in. We thus renamed our own class slightly to avoid a clash. Notice that the class statement defines our new class to be based on SimpleHTTPRequestHandler, the default.

Our do_GET() function thus replaces the default version. We test the path using the if/elif/else parts, and implement the reset/stop/restart codes as before. But now, if the supplied path is not one of these, the we use super() to run the do_GET() of the original class, so that it can implement the default behavior.

```
from http.server import HTTPServer, SimpleHTTPRequestHandler
from microbitpc import *

class SimpleHTTPRequestHandlerMB(SimpleHTTPRequestHandler):

  def do_GET(self):
    print(self.headers)
    if self.path == "/resetMB":
      self.send_response(200)
      self.end_headers()
      self.wfile.write(b'Resetting your Microbit')
      com = openMB()
      resetMB(com)
      closeMB(com)
    elif self.path == "/stopMB":
      self.send_response(200)
      self.end_headers()
      self.wfile.write(b'Stopping your Microbit')
      com = openMB()
      stopMB(com)
      closeMB(com)
    elif self.path == "/restartMB":
      self.send_response(200)
      self.end_headers()
      self.wfile.write(b'Restarting your Microbit')
      com = openMB()
      stopMB(com)
      restartMB(com)
      closeMB(com)
    else:
      super().do_GET()

httpd = HTTPServer(('localhost', 8000), SimpleHTTPRequestHandlerMB)

print("Starting server")
httpd.serve_forever()
```

You access the above using

- **http://localhost:8000/resetMB (http://localhost:8000/resetMB)**
- **http://localhost:8000/resetMB (http://localhost:8000/resetMB)**
- **http://localhost:8000/resetMB (http://localhost:8000/resetMB)**
- **http://localhost:8000/ (http://localhost:8000/)** + anything else

This code should get you started with server code.

7.