Engenharia de Serviços em Rede
Trabalho Prático Nº.3 – Serviço *Over the Top* para entrega de multimédia

José Santos, Leonardo Marreiros, and Pedro Fernandes

University of Minho, Informatic Department, 4710-057 Braga, Portugal
e-mail: {a84288, pg47398, pg47559}@alunos.uminho.pt

Resumo O seguinte trabalho realizado para a Unidade Curricular de Engenharia de Serviços de Rede tem como objetivo a criação de um protótipo de entrega de texto, áudio ou vídeo em tempo real a partir de servidor de conteúdos para N clientes. Para testar o mesmo recorrer-se-á ao emulador CORE e a várias topologias de teste. A ideia é a criação de uma rede *overlay* aplicacional formada por um conjunto de nós que podem ser usados como intermediários para reenviar dados.

Keywords: *Streaming* · Rede *Overlay* · TCP · *Multicast*

1 Introdução

Nos últimos 50 anos a Internet sofreu alterações enormes ao nível do seu paradigma. Cada vez mais são consumidos conteúdos de forma contínua e por vezes em *live*, de tipos diversificados e a todo o instante, o que contrasta com a comunicação fim-a-fim de *end system* para *end system* que se observava mais antigamente. Esta nova realidade originou grandes desafios a nível da infraestrutura IP. A solução adotada para a entrega massiva de conteúdos passa pelas *CDNs* (redes sofisticadas de entrega de conteúdos) e com serviços OTT (*Over the Top*) que são desenhados a nível aplicacional. Para o cliente final obter entregas sem perda de qualidade e em tempo real, estes serviços OTT usam uma rede *overlay* aplicacional configurada e gerida de modo a arranjar a solução para a congestão e limitação de recursos presente na rede de suporte. A *Netflix* entre outros serviços de *streaming*, formam uma rede *overlay* proprietária sobre protocolos aplicacionais (HTTP) e de transporte (TCP ou UDP), que vai correr, como o nome indica, por cima da rede IP pública. O objetivo deste trabalho é criar um protótipo de um serviço semelhante ao fornecido por estas empresas, tendo em conta uma boa qualidade de experiência do utilizador, não deixando de parte a sua eficiência e otimização de recursos.

2 Arquitetura da solução

O grupo decidiu utilizar a linguagem *Python* assim como o protocolo de transporte TCP para o envio e a receção de mensagens de controlo e de data. O

servidor inicia um servidor que escuta por pedidos de clientes e inicia também o ott, assumindo-se como *bootstrapper*. Cada ott passa por um processo de autenticação garantindo que todos os nodos vizinhos conhecem o seu id. O cliente envia um pedido de *stream* ao servidor e recebe-o pelo ott. Para recebermos vários pedidos no ott usamos *selectors*, em que sempre que um descritor de ficheiro associado ao socket diz que tem algum evento (escrita ou leitura) é provocada uma ação baseada no estado do Nodo do ott e no tipo de evento. Temos um *dispatcher* que trata de despachar os pacotes para os nodos correspondentes através do id.

3 Especificação do(s) protocolo(s)

3.1 Formato das mensagens protocolares

De modo a permitir um correto funcionamento do nosso prototipo foi necessário a troca de diversas mensagens entre os seus elementos, assim surgiram os seguintes tipos de mensagens:

- ***ACK :*** Mensagem utilizada para o envio de *Acknowledgements*. Contém no seu interior o ID do remetente.
- ***DATA :*** Mensagem utilizada para o envio de dados. Contém no seu interior o ID do remetente, os dados a enviar(Packet RTP) e um *tracker*.
- ***MESSAGE :*** Mensagem de tipo geral. Contém o ID do remetente, pode conter um *tracker* e um *timestamp*.
- ***PING :*** Mensagem utilizada para realização de um *ping*. Contém o ID do remetente e um *tracker*.
- ***SPEERS :*** Mensagem utilizada para enviar os vizinhos a um determinado nodo. Contém no seu interior o ID do remetente e os vizinhos do nó destino.

3.2 Interações

Quando um nodo se liga ao *bootstrapper*, este primeiro envia-lhe uma mensagem do tipo *ACK* contendo o seu ID. O *bootstrapper* responde-lhe mandando uma mensagem do tipo *SPEERS* contendo a lista dos vizinhos que esse nodo apresenta. O nodo responde com um *FINALACK* e o *bootstrapper* confirma que o processo de autenticação chegou ao fim. Passando ambos o estado para *CONNECTED*, se forem vizinhos.

Durante esta interação tanto o nodo como o nodo(a conectar) passam por diferentes estados. O processo de autenticação não difere sendo o nodo o *bootstrapper* ou um simples nodo na rede. A única diferença é a lista de vizinhos que apenas o *bootstrapper* manda pois apenas este os conhece.

Começando no estado *ACKRECEIVING*, o nodo que recebe a ligação após receber um *ACK* com o ID do ott do nodo que se está a ligar, o seu estado passa a ser *SPEERS*. De seguida envia a sua lista de vizinhos, e o seu estado passa para *WACK*, aqui ocorre a confirmação de que ambas as partes têm o ID correto , e passa então para o estado *CONNECTED* caso sejam vizinhos , *OFFLINE* caso

não o sejam (Isto só acontece no caso do *bootstrapper* , pois só o bootstrapper se liga a nodos na rede que não sejam vizinhos).

Já o nodo que inicia a conexão passa também por uma sequência de estados. Começando no estado *ACK_SENDING* que envia o *ACK* com o ID do ott passando o seu estado para *WPEERS*, seguido da receção da lista de vizinhos(caso seja o *bootstrapper*) e passa a *FACK* , enviado um último *acknowledgment*.

Quanto à *stream*, o servidor recebe um pedido de *stream* de um cliente e envia o pacote para o ott com o respetivo *Tracker*. O ott a seguir trata de enviar para o destino pretendido.

Quando um ott se desliga, avisa os vizinhos e o servidor de que se vai desligar.

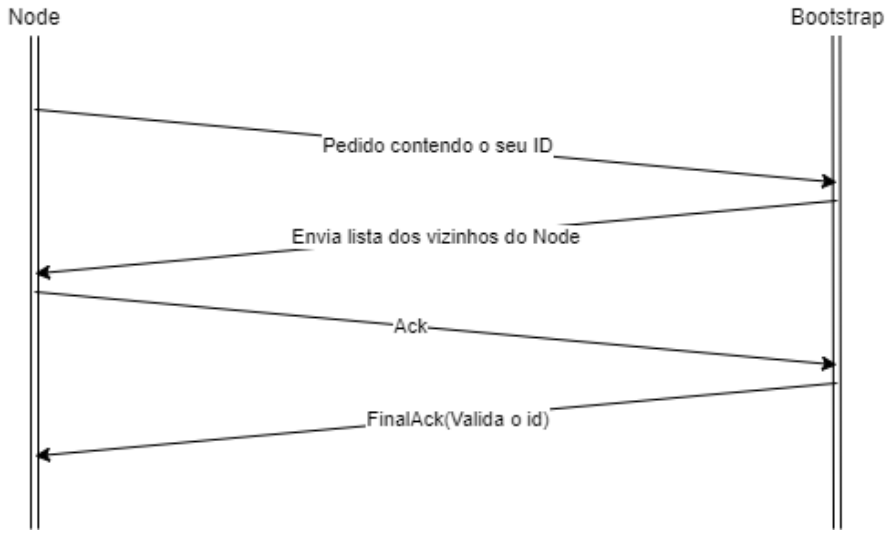


Figura 1. Estabelecimento de ligação do nodo ao *bootstrap*

4 Implementação

4.1 Construção da topologia *overlay*

Optou-se por implementar a **estratégia 2** nesta etapa. Existe um controlador (*bootstrapper*) que tem conhecimento de todo o *overlay* a partir de um ficheiro JSON com a sua configuração. Neste ficheiro estão contidos todos os nodos do *overlay* e os seus vizinhos.

Ex:

```
"10.0.1.2": {
  "neighbors": [
    "10.0.0.1",
```

```
      "10.0.3.20"
    ]
}
```

4.2 Construção das rotas para os fluxos

Construção de rotas

A partir da configuração do ficheiro JSON com a descrição da topologia, foi criado um grafo com os nodos do *overlay* e vizinhos como arestas. A partir deste grafo foi criada uma função `shortest_path` que dado um grafo e um nodo de inicio e de fim, calcula o caminho mais curto utilizando um algoritmo *breath first search*. É portanto utilizada a métrica de saltos para decidir o melhor caminho.

Rotas *Multicast*

Para determinar o melhor caminho *multicast*, isto é, o caminho para múltiplos destinos onde as mensagens passam apenas uma vez por *link* e estas apenas são duplicadas quando o *link* para os destinos diverge. Foram implementadas as funções `multicast_path_list` e `multicast_path` que, dado um nodo de início e um ou mais nodos de destino, criam este caminho. A função `multicast_path_list` devolve uma lista com os caminhos mais curtos entre o nodo inicial e cada um dos destinos. Já a função `multicast_path` utilizando esta lista, encontra o caminho que fica mais tempo unido para evitar o envio de informação duplicada por caminhos iguais.

Ex:

```
multicast_path_list("10.0.0.10",["10.0.4.20","10.0.3.20"])
> [['10.0.0.10', '10.0.0.1', '10.0.2.2', '10.0.4.20'],
  ['10.0.0.10', '10.0.0.1', '10.0.1.2', '10.0.3.20']]
multicast_path(pathlist)
> ['10.0.0.10', '10.0.0.1', [['10.0.2.2', '10.0.4.20'],
  ['10.0.1.2', '10.0.3.20']]]
```

Tracker

Para o nosso ott saber o percurso a percorrer implementamos uma classe *Tracker* que todas as mensagens têm, excepto as de autenticação. Esta classe tem na sua constituição o destino ou no caso do *multicast* , os destinos , o número de saltos dado, um conjunto de nodos por onde a mensagem passou e uma lista de nodos ordenada com o ID do ott de origem da mensagem até ao ott destino. Com esta lista e com o número de saltos dado , sabemos onde estamos e qual é o próximo nodo a quem o ott tem de enviar a mensagem. Usando as rotas *multicast* compomos a nossa lista e sabemos que quando o nosso próximo canal é uma lista , ou seja tem múltiplos paths, então temos de decompor o *multicast* e enviar por *singlecast* aos destinatários.

Caso o *tracker* tenha no *path* -1 então o ott trata de enviar para todos os nodos a mensagem, evitando enviar por onde a mensagem já passou.

4.3 Teste dos percursos no *overlay*

Para testar os percursos no *overlay*, enviamos de 1 em 1 segundo um *ping* do servidor até ao cliente pela rede ott, apresentando a informação por onde ele passa e para onde vamos enviar em cada nodo do ott. Chegando ao nodo destino(cliente) imprime a latência.

4.4 *Streaming*

O servidor escuta por pedidos de *stream* do cliente. Quando um cliente se conecta o *server* atualiza o *path* para onde manda a *stream* e envia para o ott para o ott dar *dispatch* encapsulando o pacote RTP num *DataMessage*. Quando o cliente se desliga , avisa o *server* e o *server* desliga a *stream* para aquele cliente em questão ou desliga a *stream* se este for o único cliente dele. Quando a *stream* chega ao fim , começamos o *VideoStream* de novo e enviamos novamente para os clientes ligados ao *server*.

4.5 Bibliotecas de Funções

***socket*:** permite acesso à interface de sockets BSD.

***interfaces*, *ifaddresses*, AF_INET de *nefitaces*:** biblioteca de *Python* construída para enumerar as interfaces de de rede numa máquina local.

- ***interfaces***: lista os identificadores das interfaces de uma máquina.
- ***ifaddresses***: fornece o endereço de uma interface em particular.
- **AF_INET**: coloca os endereços no formato AF_INET.

***threading*:** constrói interfaces de *threading* alto nível em cima do módulo *_thread* baixo nível.

***ThreadPoolExecutor* de *concurrent.futures*:** fornece uma interface alto-nível para a execução de *callables* assincronamente.

- ***ThreadPoolExecutor***: é um subclasse executora que utiliza uma *pool* de *threads* para executar chamadas assincronamente.

***time*:** fornece diversas funções relacionadas com tempo.

***datetime*:** fornece classes para manipular datas e tempo.

***hashlib*:** implementa uma interface que contém diversos algoritmos de resumo de mensagens e*hashes* seguros.

***pickle*:** implementa protocolos binários para a serialização e deserialização de objetos python.

***select*:** dá acesso a funções *select()* e *poll()* disponíveis na maioria dos sistemas operativos.

***selectors*:** permite multiplexagem alto nível e eficiente, construído sobre as primitivas de módulo selecionados.

***json*:** biblioteca utilizada para manipular ficheiros JSON.

***enum*:** classe para a criação de enumerações.

***logging*:** biblioteca que fornece uma *framework* flexível para emissão de *log messages* por parte de programas *Python*.

5 Testes e resultados

Para alterar a rede em questão modificamos no common.py a linha referente ao path do json(pathToNetworkConfig)

5.1 Cenário 2

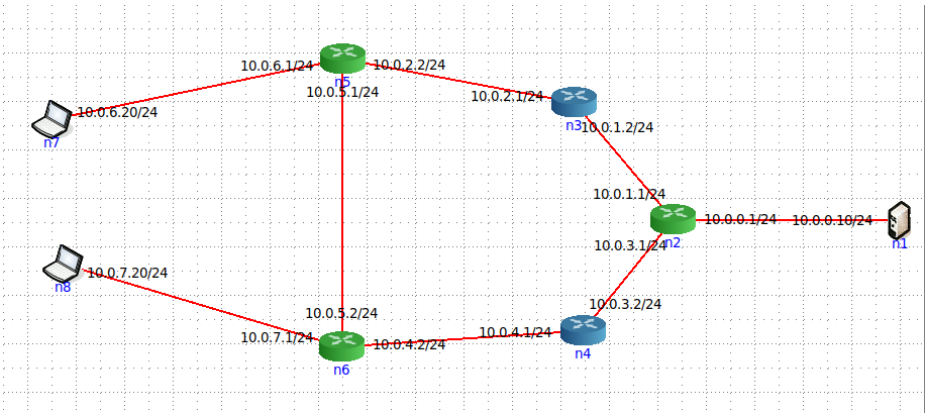
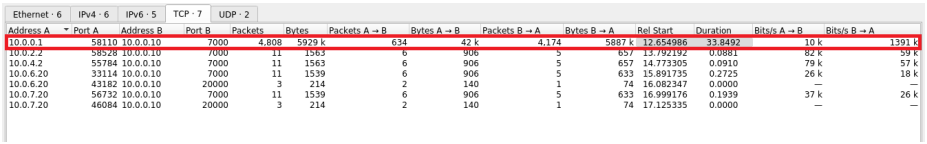


Figura 2. Topologia do cenário 2 - Nodos do *overlay* a verde

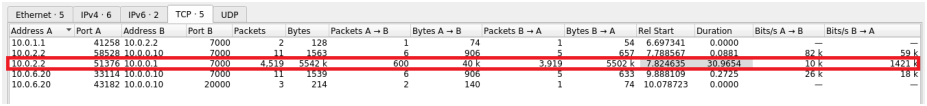
Para testar este cenário, foi executado o *Wireshark* em cada uma das interfaces do *router* do nodo n2 enquanto se pediu o vídeo em cada um dos clientes. Num

cenário sem *multicast*, a quantidade de *bytes* recebida pela interface 0 (10.0.0.1) seria o dobro daquela que seria enviada pelas interfaces 1 e 2. Assim sendo, com *multicast*, esperamos ver uma quantidade aproximadamente igual de *bytes* enviados e recebidos.



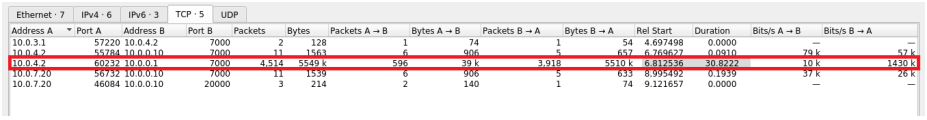
Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
10.0.0.1	58110	10.0.0.10	7000	4,808	5929 k	634	42 k	4,174	5887 k	12.654986	33.8492	10 k	1391 k
10.0.2.2	58528	10.0.0.10	7000	11	1563	6	906	5	657	13.792192	0.0881	82 k	59 k
10.0.4.2	55784	10.0.0.10	7000	11	1563	6	906	5	657	14.773305	0.0910	79 k	57 k
10.0.6.20	33114	10.0.0.10	7000	11	1539	6	906	5	633	15.891735	0.2725	26 k	18 k
10.0.6.20	43182	10.0.0.10	20000	3	214	2	140	1	74	16.082347	0.0000	—	—
10.0.7.20	56732	10.0.0.10	7000	11	1539	6	906	5	633	16.999176	0.1939	37 k	26 k
10.0.7.20	46084	10.0.0.10	20000	3	214	2	140	1	74	17.125335	0.0000	—	—

Figura 3. Quantidade de *bytes* recebidos pela interface 10.0.0.1 no nodo n2



Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
10.0.1.1	41258	10.0.2.2	7000	2	128	1	74	1	54	6.697341	0.0000	—	—
10.0.2.2	58528	10.0.0.10	7000	11	1563	6	906	5	657	7.788567	0.0881	82 k	59 k
10.0.2.2	51376	10.0.0.1	7000	4,519	5542 k	600	40 k	3,919	5502 k	7.824635	30.9654	10 k	1421 k
10.0.6.20	33114	10.0.0.10	7000	11	1539	6	906	5	633	9.888109	0.2725	26 k	18 k
10.0.6.20	43182	10.0.0.10	20000	3	214	2	140	1	74	10.078723	0.0000	—	—

Figura 4. Quantidade de *bytes* enviados pela interface 10.0.1.1 no nodo n2



Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
10.0.3.1	57220	10.0.4.2	7000	2	128	1	74	1	54	4.697498	0.0000	—	—
10.0.4.2	55784	10.0.0.10	7000	11	1563	6	906	5	657	6.769627	0.0910	79 k	57 k
10.0.4.2	60232	10.0.0.1	7000	4,514	5549 k	596	39 k	3,918	5510 k	6.812536	30.8222	10 k	1430 k
10.0.7.20	56732	10.0.0.10	7000	11	1539	6	906	5	633	8.995492	0.1939	37 k	26 k
10.0.7.20	46084	10.0.0.10	20000	3	214	2	140	1	74	9.121657	0.0000	—	—

Figura 5. Quantidade de *bytes* enviados pela interface 10.0.3.1 no nodo n2

A quantidade de *bytes* recebida pelo servidor na interface 0 do *router* n2 (10.0.0.1) (5887k) foi aproximadamente igual à quantidade de *bytes* enviada pelo *router* nas interfaces 1 (10.0.1.1) e 2 (10.0.3.1) (5502k e 5510k respetivamente) pelo que podemos confirmar que o *multicast* está a funcionar como esperado. A maior quantidade de *bytes* enviado pelo servidor à interface 0 é devido a mensagens de controlo que são enviadas principalmente no início da conexão dos nodos *overlay* à rede.

5.2 Cenário 2

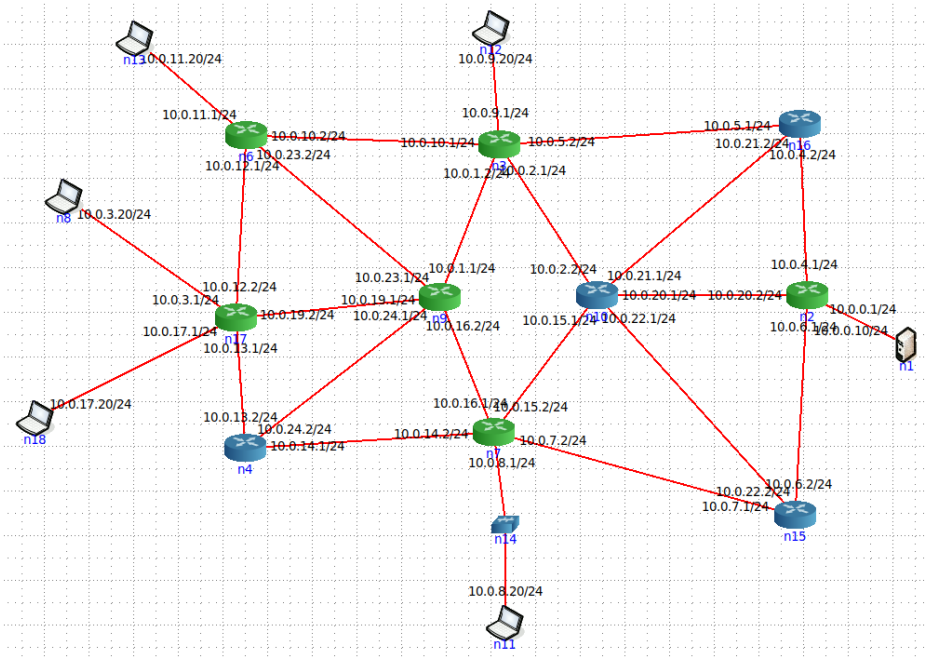


Figura 6. Topologia do cenário 2 - Nodos do *overlay* a verde

Neste cenário procuramos testar a funcionalidade de desligar um nodo do *overlay*. Seria de esperar que fosse encontrado um caminho alternativo de modo a preservar o serviço de *streaming* mesmo quando algum nodo falha.

Para fins de teste foram ligados os otts n2,n3, n6 e n17 de modo a chegar ao *laptop* n8 tendo apenas um caminho possível. Em seguida o *laptop* n8 pede a *stream* ao servidor. Posteriormente liga-se o nodo n9 e em seguida quebra-se o caminho inicial desligando o n6.A *stream* passa a ir por esse nodo permitindo ao cliente continuar a receber a *stream* enquanto desejar.

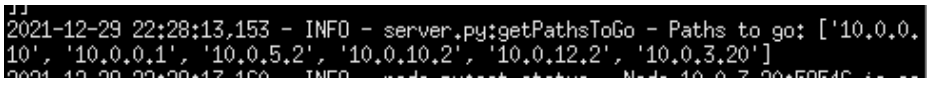


Figura 7. Caminho com o nodo n17 ligado e o nodo n9 desligado

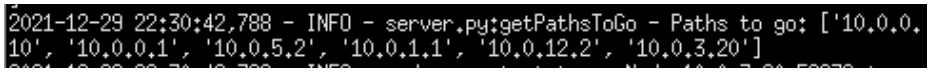


Figura 8. Caminho com o nodo n17 desligado e o nodo n9 ligado

Como pudemos ver o sistema encontrou um caminho alternativo que permitiu que o cliente em causa pudesse continuar a usufruir do serviço de *streaming*.

6 Conclusões e trabalho futuro

Dado por concluído o trabalho entendemos que realizamos com sucesso e de forma eficaz a tarefa proposta relativa à implementação de um protótipo de *streaming* assim como todas as suas dependências. O protótipo desenvolvido foi realizado etapa a etapa, tendo sempre sido tomadas as decisões de forma consciente e ponderada.

Algo que poderia ter sido feito de forma diferente é, por exemplo, a construção do caminho *multicast*. Ao invés de procurar apenas o caminho mais curto para os destinos, poderia ter sido tomado mais partido do *multicast* e construído o caminho com mais ramos em comum, mesmo não sendo o mais curto para todos os destinos.

No futuro poderíamos implementar funcionalidades extra como a implementação do *play* e *pause* da *streaming* assim como cada cliente poder escolher o ficheiro que pretende receber. Para além disso poderiam ser realizadas alterações à camada *over the top*, como por exemplo a descoberta autónoma por parte dos nodos dos caminhos por onde deve enviar pacotes, assim como a descoberta dos seus *peers* vizinhos.

Em geral, consideramos que o balanço do trabalho é positivo, as dificuldades sentidas foram superadas e os requisitos básicos propostos cumpridos.

Referências

1. Kurose, James, Ross, Keith: Computer Networks - A Top-Down Approach. 7th edition, 2017
2. Socket Programming in Python, https://realpython.com/python-sockets/. Last accessed 9 Dec 2021
3. pickle - Python object serialization, https://docs.python.org/3/library/pickle.html. Last accessed 12 Dec 2021
4. threading - Thread-based parallelism, https://docs.python.org/3/library/threading.html. Last accessed 11 Dec 2021
5. Python JSON, https://www.w3schools.com/python/python_json.asp. Last accessed 12 Dec 2021