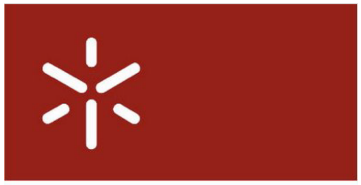

Universidade do Minho

Universidade do Minho

Arquiteturas Emergentes de Redes

Implementação de um jogo distribuído numa Rede Veicular (Rede Oportunista/Tolerante a Atrasos)

Grupo 1

Maria João Moreira (A89540)

José (A84288)

junho de 2022

Conteúdo

Capítulo 1

Introdução

No âmbito da unidade curricular de Arquiteturas Emergentes de Redes, foi pedido que fosse implementada uma aplicação que permita a conexão de vários clientes através de uma rede de comunicações.
Num primeiro momento, o objetivo consistiu em desenvolver a aplicação usando uma rede IPv6 infraestruturada. De seguida, o jogo desenvolvido deveria ser implementado para uma rede veicular ad-hoc, em configuração de rede tolerante a atrasos (DTN, Delay Tolerant Network).
Havia ainda a opção adicional de implementar o paradigma NDN (Named Data Networks).

Capítulo 2

Aplicação

A aplicação escolhida foi um jogo similar ao famoso Agar.io onde num mapa delimitado, os jogadores têm como objetivo ficar o maior possivel, para tal podem comer bolinhas que vão surgindo no mapa ou podem comer outros jogadores. Quando dois players se tocam o player de maior massa come o player de menor massa, adicionando a si mesmo a massa do player derrotado. O vencedor é aquele que conseguir ficar maior e por consequencia eliminar todos os outros players.
O desenvolvimento do jogo foi realizado através da framework do Python, Pygame No nosso caso apenas dá para mover e comer as bolinhas pequenas crescendo de tamanho.

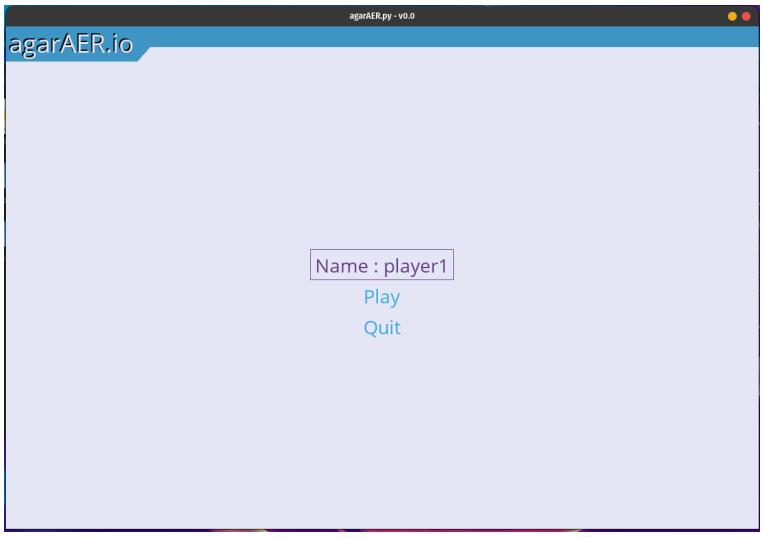

Figura 2.1: Menu Inicial

Resumidamente, o jogo possui entidades do tipo Drawable, que são todos elementos possíveis de desenhar no nosso plano de jogo.

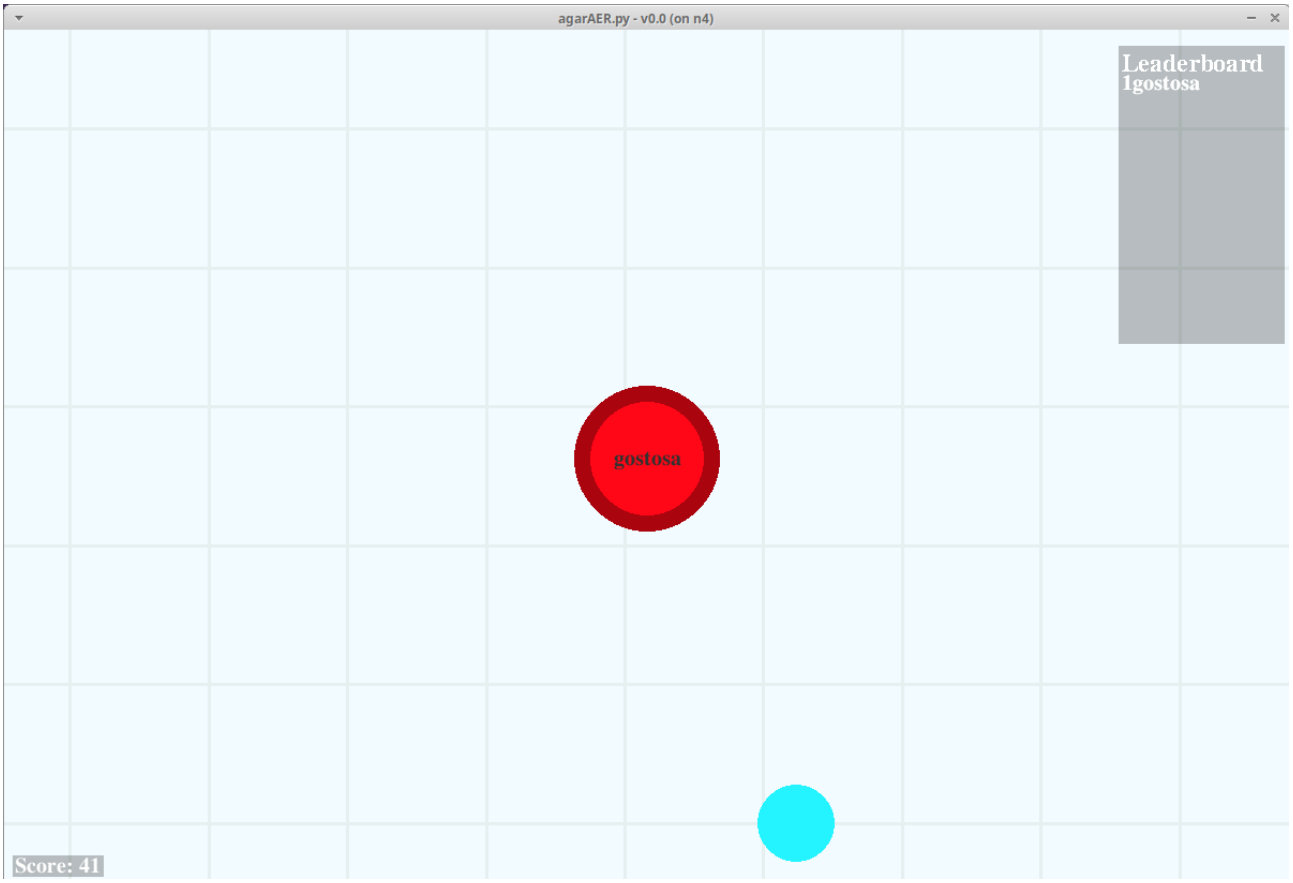

Figura 2.2: Jogo a decorrer

Capítulo 3

Cliente-Servidor

O servidor será responsavél por permitir a conexão de multiplos clientes que iniciem o jogo, este gere a informação recebida a partir dos pacotes enviados pelo cliente, aplica as mudanças de acordo com essas informações e envia pacotes com a informação inerente ao jogo atualizada para todos os clientes, de maneira a estes possuírem o mesmo estado de jogo. Esta troca de pacotes entre cliente e servidor está constantemente a acontecer através de multicast.
O protocolo escolhido para implementar o multicast foi o pim6sd que se configura automaticamente para realizar fowarding em todas as interfaces que suportam multicast.
A comunicação entre cliente servidor dá-se através dos seguintes passos:

1. Servidor liga-se e fica à escuta

```
def start_listening(self):
    self.poll.register(self.newconn_watcher.get_watch(), select.POLLIN)
    threading.Thread(target=self.checkPlayerStatus).start()
    try:
        while True:
            sleep(0.01)
            events = self.poll.poll(1)
            # For each new event, dispatch to its handler
            for key, event in events:
                if key == self.newconn_watcher.get_watch():
                    self.handle_new_connection(key,event)
                else:
                    self.threadpool.submit(self.handler, key, event)
    finally:
        pass
```

2. Cliente liga pela primeira vez e envia para o servidor um pacote com a sua porta, o seu nome e o seu id, a este pacote chamamos AuthenticationRequest. Para tal o servidor executa a função handle_new_connection() definida abaixo. Nesta função o servidor verifica se recebeu uma AuthenticationRequest e se este for o caso, adiciona o novo player ao jogo e envia a configuração através da função sendConfig(). De seguida vai ficar à escuta de pacotes desse player na porta que este está conectado através da função listenForClientUpdates().
   Por fim se o jogo ainda não tiver sido iniciado, ou seja se este é o primeiro cliente que se está a conectar, então server inicia o jogo, porém se o jogo já tiver sido iniciado então ele adiciona o novo jogador ao jogo.

```
def handle_new_connection(self, key, event):
    data ,addr = self.newconn_watcher.retrieveMessage()
    authrequest = pickle.loads(data)
    if authrequest.type != MessageType.AUTHENTICATION_REQUEST:
        return
    logging.log(logging.INFO, "New connection from %s" % str(addr))

    p = self.game.add_player(addr,
                             authrequest.get_id(),
                             authrequest.get_name(),
                             authrequest.get_port())
    self.threadpool.submit(self.sendConfig,p,authrequest.get_port())
    self.listenForClientUpdates(authrequest.get_port())
    if not self.gameStarted:
        self.initGame()
    else:
        self.game.add_to_newplayers(p, 20)
```

3. Servidor envia para cada cliente que se ligou pela primeira vez através da porta que o mesmo especificou informações sobre o próprio jogador, são elas: o seu id, a sua cor, a sua massa e a sua posição inicial, bem como a informação sobre o jogo em geral e os jogadores que ja estiverem conectados.

4. Cliente a cada frame envia mensagem com id e o seu estado (a sua massa, a sua rotação e as bolinhas que já comeu)

5. Servidor calcula o movimento do cliente e dá update ao estado geral do jogo e envia para todos os clientes conectados o estado do jogo mais atualizado.

Acontece que os pacotes com a informação sobre a configuração do jogo que o servidor transmite a todos os jogadores, possui um tamanho maior do que 1500 bytes , o que a excede o tamanho suportado pelo UDP, portanto decidimos dividir a informação destes pacotes em chunks para garantir que toda a informação é transmitida na sua totalidade aos clientes, uma vez que o UDP não garante a entrega de pacotes que excedam a capacidade máxima.

Abaixo está definida a função que cria a mensagem com toda a configuração a ser transmitida aos clientes, realiza a sua divisão em chunks e transmite-os.

```
def sendConfig(self,p,port):
    # while not p.get_acceptconf_status():
     sleep(1)
     tmpsock = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
     tmpsock.setsockopt(socket.IPPROTO_IPV6, socket.IPV6_MULTICAST_HOPS ,64)
     tmpsock.setsockopt(socket.IPPROTO_IPV6, socket.IP_MULTICAST_TTL ,64)
     msgToSend = AuthenticationResponse(p.get_id(),
                                        self.createConfigForNewPlayers(p))
     finalmsg = pickle.dumps(msgToSend)
     if len(finalmsg) > 1200:
         res = self.createConfigInPacketSize(p,msgToSend.get_config())
         for i in range(len(res)):
             tmpauthpck = res[i]
             tmpauthpck.set_last_packet_no(len(res)-1)
             tmpauthpck.set_packet_no(i)
             tmpauthpckpick = pickle.dumps(tmpauthpck)
             tmpsock.sendto(tmpauthpckpick, (self.group_addr, port))
             sleep(0.01)
     else:
         tmpsock.sendto(finalmsg, (self.group_addr, port))
```

Capítulo 4

Primeira Fase - Rede IPv6 infraestruturada

Para a rede ipv6 infraestrutrada temos nodos estáticos constantemente ligados, por isso apenas precisamos de lidar com a forma como os pacotes seriam transmitidos, e para tal, como menciondo acima, utilizamos o protocolo de routing multicast, pim6sd .
Utilizamos multicast para descobrir os nomes e endereços dos nodos conectados e, também para transmitir toda a informação do jogo, isto tem a vantagem de transmitir simultaneamente para múltiplos destinatários o estado do jogo.
Os routers que estão no meio da rede overlay são então iniciados com o protocolo de routing multicast.

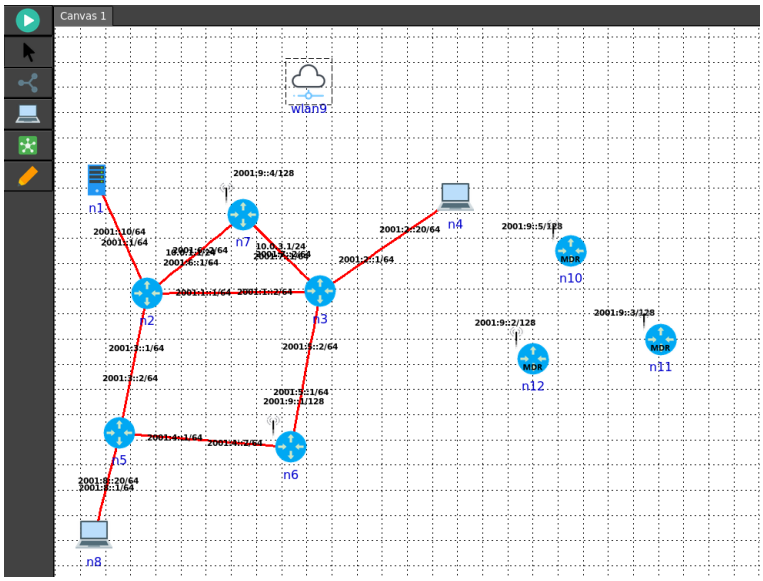

Figura 4.1: Topologia

Capítulo 5

Segunda Fase - Rede ad-hoc veicular

No contexto de uma rede veicular, lidamos com clientes que não estão estaticos, o movimento dos mesmos por vezes pode fazer com que estes percam a conexão à rede. É em cenários como este que surge a necessidade de implementar uma rede tolerante a falhas (Delay Tolerant Network - DTN), ou seja uma rede que consiga lidar com falhas de maneira a fazer com que estas falhas impactem menor numero de dispositivos possível.
Como visto anteriormente, os clientes estão sempre a trocar mensagens relevantes para o jogo com o servidor, o jogo por sua vez é impactado por todas as ações que os clientes realizem, por exemplo se um jogador se deslocar numa dada direção os outros jogadores necessitam que a posição deste chegue até eles.
Devido ao facto do jogo depender da junção de todas as atualizações que vão sendo feitas à medida que os clientes jogam e consequentemente à medida que os pacotes com estas atualizações são transmitidos para o servidor, usando um cliente se desconecta o servidor não recebe os pacotes provenientes desse nó, a este fenomeno dá-se o nome de falta.
Para contornar esta situação que é muito comum em redes veicular, devido ao facto de haver movimento, não se consegue garantir uma conexão estavél, o nó desconectado procura por vizinhos que consigam estabelecer conexão, e envia e recebe através destes vizinhos os pacotes destinados para ele.
Claro que este processo por causar algum delay no envio de pacotes, e esse delay pode tornar-se ainda mais evidente em jogos como este, emque a cada frame o servidor recebe e envia muitos pacotes, uma vez que é um jogo está constantemente a sofrer alterações no seu estado. Portanto é natural que no lado dos clientes que possuem uma conexão estavél se verifique que existe algum delay nas atualizações provocadas pelo nó que está a utilizar do modulo DTN. Tendo isso em conta foi necessário encontrar uma maneira da informação sair e chegar aos nós móveis,sendo assim, desenvolvemos um módulo DTN para lidar com redes multicast.

5.1 Implementação

O nosso módulo DTN funciona como um daemon, à parte da nossa aplicação. Tem um multicast sniffer que capta os pacotes multicast e os armazena numa cache. Toda a nossa base de informação é baseado no digest do payload do pacote, este ajuda a verificar se já temos o pacote. A nossa cache *storeService* foi pensada num género de uma loja. Temos prateleiras (grupos multicast) que são depois uma ajuda para os nós que chegam e pedem os dados de um certo grupo. Os peers têm um módulo de discovery em que basicamente estão de x em x tempo (este tempo é baseado numa distribuição uniforme) a enviar pacotes para tentarem descobrir possivéis vizinhos por onde poderão transmitir os seus pacotes de forma a tentar evitar que vários peers enviem a mensagem de hello ao mesmo tempo. Esta mensagem de hello contém informação se o nó é um nó do overlay e o tempo médio de passagem no overlay pelo peer. De seguida , o nó usa o nosso algorítmo de forwarding em que basicamente preenche uma fila de pacotes tendo em conta o próximo salto, calculado pelo nosso algorítmo, entregando-os assim ao próximo salto. O nó ao passar perto de outro, faz um pedido de dados um pedido de dados a esse vizinho *RequestService* , pedindo dados que vieram da rede fixa. Por fim para entregar tanto à rede fixa, como à aplicação dos nós , temos um serviço *DeliveryService* que trata de enviar os dados para os grupos de multicast de onde vieram, fazendo assim com que a aplicação os capte.

Como esta é uma estrutura multicast,imaginemos um cenário em que temos um Peer 1 e este se desconectou, este vai procurar um vizinho. Caso encontre um vizinho, chamemos-lhe Peer 2,próximo salto do Peer 1 irá ser o vizinho Peer 2, é então enviada uma mensagem *ForwardMessage* ao Peer 2 que faz com que este passe a receber os pacotes multicast da aplicação provenientes da aplicação do Peer 1. Desta forma, o Peer 1 conseguiu transmitir os pacotes, e agora cabe ao Peer 2 retransmitir os pacotes a um novo vizinho porém excluindo o Peer 1 como possivel vizinho uma vez que este foi quem lhe transmitiu a *ForwardMessage*.

Já na rede fixa o peer que recebe os pacotes provenientes do ultimo salto feito, despacha os pacotes que tem por todas as portas e trata de escutar os grupos multicast para onde nós enviamos dados, à espera de receber agora do servidor pacotes com as devidas atualizações do jogo.

Por fim o este peer da rede fixa, armazena numa espécie de cache os pacotes destinados aos nós da rede veicular, e quando estes nós passarem perto desse peer da rede fixa, irão receber os pacotes a si destinados. Claro que caso estes nós demorem a passar perto do peer que contém o pacote com as devidas atualizações, o jogo para estes clientes irá demorar mais a refletir essas atualizações.

5.2 Algoritmo de forwarding

Para decidirmos o nosso próximo salto, contamos com informação propagada pela rede. A decisão é tomada tendo em conta o número de vezes que passamos por um nó da rede fixa e o tempo médio para voltar a passar. Além disso , para cada nó guardamos também o número de vezes que passamos por um vizinho e o tempo médio de passagem. Conseguimos assim ter noção distância à rede fixa ou a um vizinho e desta forma podemos tomar a decisão sobre quem irá ser responsavél por ser o próximo salto e tratar da distribuição dos pacotes.

5.3 Guia de utilização

Para usar o Node DTN é usar o comando python3 DTN.py –i (Interface da aplicação) –o (Interface da rede fixa pertencente à rede multicast) A opção –o é opcional e indica se o nó é um nó overlay ou não.

Para correr o pim6sd é necessário nos nós da rede fixa (os que fazem ponte entre a rede fixa e a rede móvel, não pertencem) correr o comando pim6sd -f (ficheiro de configuração) [pim6sd.conf]/[2.conf]

Para correr o servidor é só correr python3 runServer.py.

E para correr o nosso cliente de jogo é correr python3 agarAER.py

Capítulo 6

Conclusão e trabalho futuro

No âmbito da unidade curricular, colocamos em prática as noções adquiridas acerca de redes ad-hoc veiculares que utilizam dos módulos DTN para lidar com possivéis falhas na rede. Exercitamos consequentemente também, estruturas cliente-servidor e tiramos proveito de protolocos multicast.
Com este trabalho pretendia-se estudar a troca rápida e constante de informações em redes veiculares. Com isto em mente, desenvolvemos um jogo que exigia múltiplas trocas de informação. Através disto , conseguimos analisar e compreender a maneira como as redes veiculares têm de lidar e propagar a informação. Como a nossa aplicação é um jogo que se quer que tenha muita pouca latência, não é a aplicação mais fácil para implementar num cenário como este. Com isto , concluímos que foi um trabalho que nos deu uma experiência positiva e enriquecedora do que é trabalhar com DTN's.