Implementação da comunicação direta entre máquinas via socket

Visão geral

Para melhor entender o funcionamento da camada de transporte, com a ajuda desse tutorial, você elaborará uma comunicação entre duas ou mais máquinas que compõem a estrutura do *testbed* **FIBRE**. Para isso, fará um breve estudo dos dois principais protocolos de transporte utilizados na Internet, **UDP** e **TCP**, em que implementará uma troca de mensagens através do uso desses dois protocolos. Após tal estudo, usará um destes protocolos e suas habilidades de programação para desenvolver o sistema de monitoramento de recursos computacionais.

Competências desenvolvidas

1. Acesso e uso do *testbed* **FIBRE**, através da criação de experimentos em ambientes virtualizados e alocação de máquinas virtuais em locais espalhados pelo Brasil na plataforma **OMF6**.
2. Estudo prático dos protocolos de transporte **UDP** e **TCP**.
3. Fabricação de softwares para utilização dos protocolos da camada de transporte na troca simples de mensagens entre uma máquina servidor e uma máquina cliente.
4. Reelaboração dos softwares de troca de mensagens a fim de elaborar um monitor de recursos computacionais disponíveis.

Licenciamento

Esse material foi desenvolvido por **Vinícius Julião Ramos** - Universidade Federal de Minas Gerais, através de recursos disponibilizados através da **Chamada Interna de Tutoriais FIBRE**.

LISTA DE FIGURAS

Figura 1 – Tabela das camadas de protocolos segundo dois modelos e a divisão entre parte inferior e superior. 2
Figura 2 – Topologia em estrela de uma rede de grande área (WAN) 4
Figura 3 – Empilhamento e desempilhamento dos protocolos na arquitetura TCP/IP 5
Figura 4 – Ecaminhamento e montagem do endereçamento através da multiplexação e demultiplexação. 8
Figura 5 – Exemplo de multiplexador de quatro entradas e demultiplexador de quatro saídas. 10
Figura 6 – Diagrama de comunicação paralela do TCP. 14
Figura 7 – (Comunicação Básica) My projects 16
Figura 8 – (Comunicação Básica) Detalhes do slice 17
Figura 9 – (Comunicação Básica) Endereços IP das máquinas 18
Figura 10 – (Comunicação Básica) Editor de texto para servidor UDP 19
Figura 11 – (Comunicação Básica) Código para servidor UDP 20
Figura 12 – (Comunicação Básica) Editor de texto para cliente UDP 20
Figura 13 – (Comunicação Básica) Código para cliente UDP 21
Figura 14 – (Comunicação Básica) Comando de execução do cliente UDP com servidor offline 21
Figura 15 – (Comunicação Básica) Executando cliente UDP com servidor offline 22
Figura 16 – (Comunicação Básica) Comando de execução do servidor UDP . . 22
Figura 17 – (Comunicação Básica) Execução do cliente UDP em funcionamento 22
Figura 18 – (Comunicação Básica) Execução do servidor UDP em funcionamento 23
Figura 19 – (Comunicação Básica) Comando vim para servidor TCP 23
Figura 20 – (Comunicação Básica) Código para servidor TCP 24
Figura 21 – (Comunicação Básica) Comando vim para cliente TCP 24
Figura 22 – (Comunicação Básica) Código para cliente TCP 25
Figura 23 – (Comunicação Básica) Execução do cliente TCP com servidor offline 26
Figura 24 – (Comunicação Básica) Execução do servidor TCP 26
Figura 25 – (Comunicação Básica) Execução do cliente TCP 26
Figura 26 – (Comunicação Básica) Resultado da comunicação entre cliente e servidor TCP 27
Figura 27 – (Sistema de Monitoramento) Criação do slice 30
Figura 28 – (Sistema de Monitoramento) Chamada do vim para criação do arquivo do Anexo E 30

Figura 29 – (Sistema de Monitoramento) Colagem do Anexo E no arquivo do cliente rnp 31
Figura 30 – (Sistema de Monitoramento) Chamada do vim para criação do arquivo do Anexo F na máquina rio_grande_do_sul 31
Figura 31 – (Sistema de Monitoramento) Colagem do Anexo F no arquivo do servidor rio_grande_do_sul 32
Figura 32 – (Sistema de Monitoramento) Chamada do vim para criação do arquivo do Anexo F na máquina bahia 32
Figura 33 – (Sistema de Monitoramento) Colagem do Anexo F no arquivo do servidor bahia 33
Figura 34 – (Sistema de Monitoramento) Instalação do pip3 em rio_grande_do_sul 34
Figura 35 – (Sistema de Monitoramento) Instalação da biblioteca psutil pelo pip3 em rio_grande_do_sul 34
Figura 36 – (Sistema de Monitoramento) Instalação do pip3 em bahia 34
Figura 37 – (Sistema de Monitoramento) Instalação da biblioteca psutil pelo pip3 em bahia 35
Figura 38 – (Sistema de Monitoramento) Chamada da execução do servidor rio_grande_do_sul 35
Figura 39 – (Sistema de Monitoramento) Chamada da execução do servidor bahia 35
Figura 40 – (Sistema de Monitoramento) Chamada da execução do cliente para comunicar com cada servidor em segundo plano 35
Figura 41 – (Sistema de Monitoramento) Leitura do arquivo de saída da comunicação com rio_grande_do_sul 36
Figura 42 – (Sistema de Monitoramento) Leitura do arquivo de saída da comunicação com bahia 36
Figura 43 – (Sistema de Monitoramento) Exibição da saída da comunicação entre cliente e rio_grande_do_sul 37
Figura 44 – (Sistema de Monitoramento) Exibição da saída da comunicação entre cliente e bahia 37
Figura 45 – (Sistema de Monitoramento) Finalizando execução dos clientes que estão em segundo plano de acordo com o número do processo . . . 38
Figura 46 – (Sistema de Monitoramento) Resposta do servidor rio_grande_do_sul ao encerrar o cliente 38
Figura 47 – (Sistema de Monitoramento) Resposta do servidor bahia ao encerrar o cliente 39

LISTA DE TABELAS

Tabela 1 – Tabela de Hospedeiros 9

SUMÁRIO

1 INTRODUÇÃO 1

2 CAMADA DE TRANSPORTE 2

2.1 Camadas Adjacentes 2

2.1.1 Camada de Rede 3

2.1.2 Camada de Aplicação 4

2.2 Comunicação Lógica 6

2.3 Multiplexação e Demultiplexação 8

2.3.1 Multiplexação 9

2.3.2 Demultiplexação 10

2.4 UDP - *User Datagram Protocol* 11

2.5 TCP - *Transmission Control Protocol* 12

2.5.0.1 Confiabilidade de transmissão 12

2.5.0.2 Transmissão Paralela 13

2.5.0.3 Outros controles 14

3 COMUNICAÇÃO BÁSICA 16

3.1 Criação do ambiente 16

3.2 Usando UDP 18

3.3 Usando TCP 23

3.4 Questões 27

4 SISTEMA DE MONITORAMENTO DE RECURSOS COMPUTACIONAIS 28

4.1 Implementação do Código 28

4.2 Execução 29

5 CONCLUSÃO 40

6 AGRADECIMENTOS 41

REFERÊNCIAS 42

ANEXO A – COMUNICAÇÃO BÁSICA - UDP CLIENT 43

ANEXO B – COMUNICAÇÃO BÁSICA - UDP SERVER 44

ANEXO C – COMUNICAÇÃO BÁSICA - TCP CLIENT 45

ANEXO D – COMUNICAÇÃO BÁSICA - TCP SERVER 46

ANEXO E – MONITOR DE RECURSOS - CLIENT 47

ANEXO F – MONITOR DE RECURSOS - SERVER 48

1 INTRODUÇÃO

Diante da maior rede de computadores do mundo, a Internet, é importante entender de forma prática como se dá a comunicação entre as máquinas que compõem tal rede. Para isso, é necessário a compreensão prévia a cerca da divisão das camadas da infraestrutura da Internet e cada respectivo protocolo, principalmente o protocolo **IP** e a divisão entre a camada de aplicação e a camada de transporte. No entanto, mesmo que não haja um entendimento pleno sobre estes três quesitos, este tutorial se encarregará de explicá-los brevemente, para melhorar a compreensão sobre os protocolo **TCP** e **UDP**.

A Internet não se resume à comunicação entre os *browsers* e os servidores da *Web*. A *Web* é apenas um dos serviços da rede mundial, que compõe as partes superiores na pilha de camadas dessa arquitetura, como a camada de **serviço** e **aplicação**, logo, há grande fluxo de informações que trafegam pelos roteadores "públicos"e que estão ocultos à **Web**. Um grande exemplo disso são os jogos *on line*, em que muitos destes não são acessíveis pelos navegadores, mas que usam da Internet para trocar informações entre os jogadores. Nesse cenário, os leitores entenderão como se dá tal troca de dados.

Genericamente é possível afirmar que os dois protocolos de transporte principais se diferem pela forma como enviam seus dados. Para melhor compreender a estrutura de uma conexão, consideremos a existência de um servidor, que provê o serviço, e de um cliente, que inicia as requisições ao servidor. O protocolo **TCP** orienta-se numa conexão fixa, em que para iniciar a troca de informações, é necessário, anteriormente, garantir que tal conexão está firmada. Além disso, esse protocolo, garante a integridade dos dados trafegados nessa rede. Já o **UDP** realiza a troca de informações de forma básica, em que o cliente somente envia as informações para o servidor, sem qualquer necessidade de firmar uma conexão anteriormente.

Após a compreensão teórica das duas maneiras de transportar os dados, será possível desenvolver uma comunicação entre vários clientes e um servidor, fazendo uso desses dois protocolos. A estrutura do *testbed* **FIBRE** fornecerá os recursos e um ambiente favorável para iniciar o estudo prático e experimentativo a cerca da escolha da melhor tecnologia para a criação de um **Sistema de Monitoramento Remoto de Recursos Computacionais**.

2 CAMADA DE TRANSPORTE

Dado um serviço que consista em trafegar dados por uma rede, surge a necessidade em integrar uma aplicação à um meio de tráfego. Logo, requer-se uma tecnologia que seja capaz de promover tal integração. No cenário atual das redes de computadores, uma maneira de possibilitar o funcionamento desse serviço, é através da implementação de uma estrutura de rede que seja descrita por uma pilha de camadas, em que cada nível desta pilha seja responsável por uma tarefa.

	Arquitetura TCP/IP	Modelo ISO
SUPERIOR	Aplicação	Aplicação
		Apresentação
		Sessão
INTERMEDIÁRIA	Transporte	Transporte
INFERIOR	Rede	Rede
	Enlace	Enlace
	Física	Física

Figura 1 – Tabela das camadas de protocolos segundo dois modelos e a divisão entre parte inferior e superior.

Após a análise da tabela na Figura 1, deve-se compreender as funções e características de cada camada da pilha. Na imagem, possuímos duas estruturas com muitas semelhanças, isso se deve, pelo motivo da arquitetura **TCP/IP** ser um síntese da arquitetura **ISO**, esta que é uma fórmula genérica para a criação de uma pilha de protocolos, a qual a internet está fundada e outras demais redes. Como desejamos utilizar uma linguagem alto nível e de fácil compreensão, focaremos na primeira camada inferior e a primeira camada superior ao segmento de transporte da pilha de protocolos **TCP/IP**, para isso é necessário o entendimento e a utilização de exemplos que configurem o **IP** e a camada de **aplicação**.

2.1 CAMADAS ADJACENTES

Da mesma forma como se dá o desenvolvimento de um software em módulos, submódulos, diretórios e etc, a hierarquização do trabalho de cada camada da estrutura de rede contribui para uma melhor organização do foco de implementação de um serviço. No caso hipotético, o qual desenvolver-se-a uma aplicação que troque informações pela rede, utilizando a pilha de protocolos que compõem a arquitetura **TCP/IP** para abordar o funcionamento do serviço, configura-se a camada de aplicação como

a parte superior da pilha, a camada de rede e suas subcamadas serão denominadas parte inferior. Já a parte central, ou intermediária, da pilha é composta pela camada de transporte.

A seção superior é responsável por tratar as informações, manipular dados, interagir com o usuário e etc. Já a parte inferior, por sua vez, possui o trabalho de encaminhar os dados entre os nós que compõem a rede, ou seja, responsabiliza-se por encontrar o caminho correto até chegar ao destino. Não obstante, fará-se uma explicação a cerca dos parâmetros de cada uma dessas seções da pilha de camadas para que fique claro a necessidade de um protocolo que conecte a aplicação ao encaminhamento, trabalho que é executado pela camada de transporte.

2.1.1 Camada de Rede

Como já dito anteriormente e também afirmado por Kurose & Ross (2013, p. 255) "[...]o papel da camada de rede é aparentemente simples — transportar pacotes de um hospedeiro remetente a um hospedeiro destinatário", em que "hospedeiros"são os nós finais de uma conexão. Para isso, os protocolos desta seção da pilha necessitam de implementar um conceito que envolve a criação de rotas entre tais hospedeiros, o que é chamado de **roteamento** (os algoritmos de roteamento, no geral, são interessantes de serem estudados, dada a complexidade de implementação dos códigos desse serviço).

Há diversos protocolos que são aplicados a esta camada, porém, como a abordagem deste tutorial envolve o uso da Internet como referência, usar-se-a o **IP** ***Internet Protocol*** como base para a explicação. A Internet, em sua maior parte é implementada em uma topologia do tipo **estrela** (Figura 2) em que há um elemento central que se responsabiliza por encaminhar os pacotes da rede para os hospedeiros corretos. Esse elemento central é denominado **Roteador**, porém, apesar do nome, não realiza apenas o trabalho de roteamento, mas também possui a tarefa denominada **enlaceamento** ou **repasse**. Esse serviço consiste em identificar qual será a porta de saída de determinado pacote que entra por alguma porta do roteador.

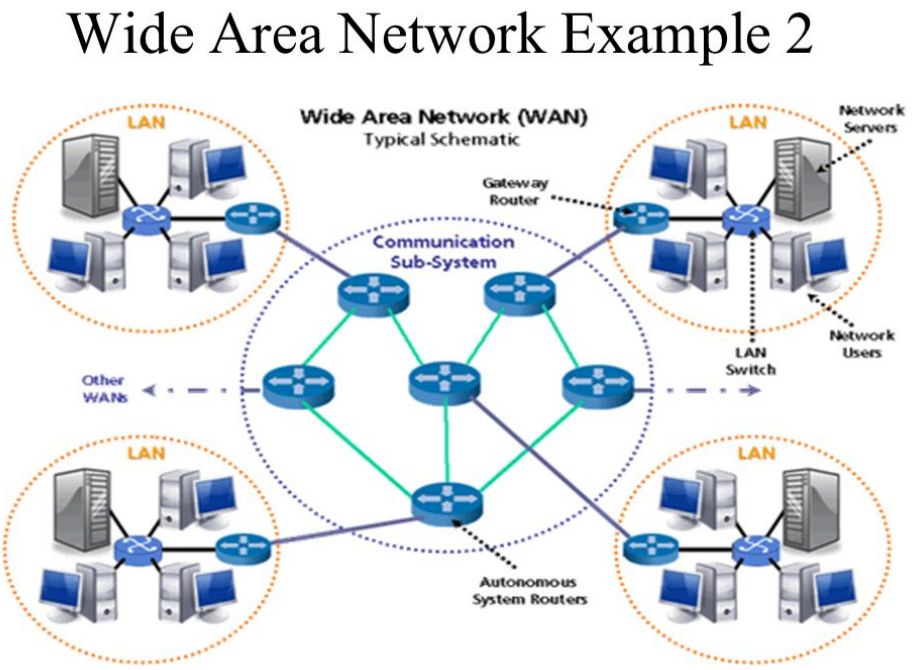


Figura 2 – Topologia em estrela de uma rede de grande área (WAN)

Fonte: Disponível em: <https://slideplayer.com/slide/7084683>. Acesso em: 5 de set. de 2019.

O trabalho de **enlaceamento** e **roteamento** só é possível por meio da identificação das portas do roteador e dos hospedeiros que compõem a rede. Tais recursos são identificados por meio do endereços **IP** que são valores únicos dentro de uma rede, ou seja, não existem dois dispositivos em uma rede com o mesmo valor de endereço **IP**, a não ser que haja a implementação de uma tradução de endereços em determinada subrede (essa prática é denominada **NAT - *Network Address Translation***). E de posse de tais endereços, os roteadores são capazes de montar tabelas de roteamento para assim aplicar os algoritmos que envolvem o processo de recebimento e entrega de pacotes.

2.1.2 Camada de Aplicação

Antes de explicar do que se trata a camada de aplicação, em uma estrutura de rede, é melhor entender o que é uma aplicação. Imagine o desenvolvimento de algum software qualquer para computador, que não necessite estar conectado a rede, por exemplo, um jogo de cartas ou um editor de texto, ou, até mesmo, apenas uma rotina no próprio computador, sem qualquer necessidade de controle por um usuário, como é o caso de alguns anti vírus - estes programas são, também, denominados aplicações. Agora, esse programa imaginário passará por uma atualização que necessite de conexão com a rede: o jogo de cartas terá uma versão *multi player on line*, o editor de texto possibilitará a edição de arquivos em algum outro servidor de forma remota,

ou o anti vírus baixará, de tempos em tempos, um pacote adicional de instruções de combate a *malwares*.

Para tornar possível as novas funções da aplicação imaginária, é necessário implementar recursos de acesso à rede. Então, torna-se cabível o uso ou o desenvolvimento de algum protocolo que conceda a comunicação entre os hospedeiros. Observando a figura 3, nota-se que em uma rotina de comunicação a aplicação localiza-se no topo da pilha, pois é o primeiro estágio do envio dos dados pelo pelo remetente e será o último estágio da leitura destes dados no hospedeiro destinatário, sendo assim, essa seção, da pilha de protocolos, é definida como a parte final da arquitetura **TCP/IP** e caracteriza-se pelo trabalho de tratar os dados da rede, transformando-os em informação, que posteriormente será utilizada por um programa específico.

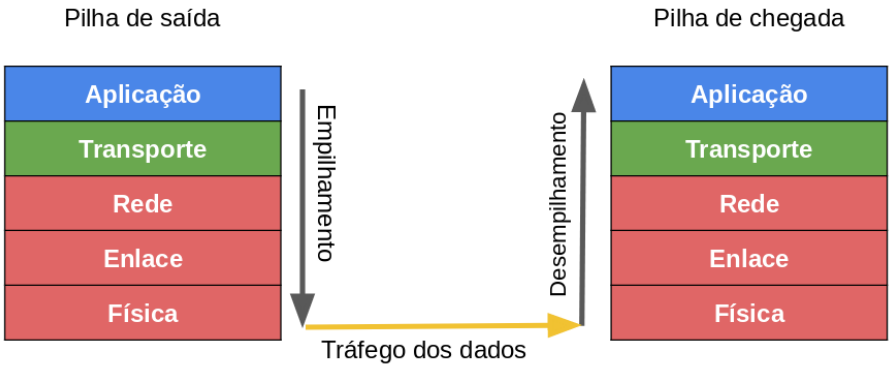


Figura 3 – Empilhamento e desempilhamento dos protocolos na arquitetura TCP/IP

Há diversos exemplos de protocolos da camada de aplicação que são utilizados por diversos programas já conhecidos. Na aplicação hipotética, do editor de texto, seria possível utilizar o protocolo ***File Transport Protocol* - FTP -** para que o programa conseguisse ler arquivos de um servidor, sobrescrevendo-os quando necessário. Também há casos em que deseja-se implementar o próprio protocolo de aplicação, vide algum quesito de segurança, por exemplo - fica claro tal necessidade tomando como referência o desenvolvimento do anti vírus, em que deseja-se aplicar uma comunicação com algum método criptográfico desconhecido por terceiros, de modo a garantir a integridade dos pacotes de instruções de segurança baixados.

Ao entender sobre a aplicação e a camada de rede, pode-se imaginar um computador em que há mais de uma aplicação, que utilize a rede, em execução. Assim, sabe-se que a camada de aplicação tratará os dados, transformando-os em informações entendíveis para os programas, e sabe-se, ainda que a camada de rede se encarregará de encaminhar da melhor maneira estes dados até os hospedeiros de destino. Mas ainda não foi dito como se dá a parte em que se define o estabelecimento de uma conexão, ou como se dá a entrega dos pacotes de dados para duas aplicações

diferentes em execução no mesmo computador. Para essas, e outras tarefas existe a **Camada de Transporte**, e as principais funções e características serão explicadas a seguir.

2.2 COMUNICAÇÃO LÓGICA

Após entender que a tarefa de encaminhar os pacotes ao longo da infraestrutura da rede cabe à seção inferior da pilha **TCP/IP**, e o serviço final de execuções dos programas é de responsabilidade da parte superior, necessita-se conhecer qual o trabalho executado pela a camada ligante dessas duas seções da extremidade da pilha de protocolos. Uma vez que a camada de rede é implementada, não somente nos roteadores (nós intermediários), mas, também, nos hospedeiros (nós finais da rede), os protocolos dessa seção fazem com que os pacotes sejam entregues de um destinatário até um remetente, entretanto cabe o serviço de identificar o endereço **IP** em que determinada aplicação deseja enviar e receber informações.

O trabalho de identificar os endereços em que determinada aplicação deseja trocar informações, é uma das tarefas da camada de transporte. Isso faz com que neste ponto da pilha em diante, todos os protocolos das camadas sejam implementados apenas nos hospedeiros, caracterizando uma comunicação fim a fim, ou seja, a camada de transporte e as camadas superiores são alienadas à forma como a informação trafega pelos nós intermediários da rede e se preocupam apenas com o recebimento ou não recebimento de determinado pacote de sua propriedade. Dessa maneira, essa camada intermediária realiza uma comunicação lógica entre os hospedeiros, através de algoritmos responsáveis pelo firmamento, ou não, de uma conexão, pela garantia de entrega de pacotes, pela abertura de um *listener* (socket) que permita que uma conexão seja criada entre dois pontos. Sendo assim, na pilha de protocolos, podemos caracterizar a camada de transporte como a responsável por propor a conexão entre dois hospedeiros.

Alguns exemplos da comunicação lógica podem ser descritos pelos seguintes pontos:

1. **Transmissão de vídeo ao vivo**: Ao imaginar a transmissão de imagens em tempo real, podemos tomar como exemplo a televisão. Numa transmissão de TV analógica (mesmo que não seja ao vivo) a comunicação é do tipo *simplex*, em que apenas um dos dispositivos transmite a informação, e se dá de tal forma em que quando há alguma interferência, impedindo que o sinal chegue com perfeição aos receptores, não existe o tratamento de erros de transmissão para validar se os dados foram entregues ou não, e nem mesmo há uma correção de erros

para deixar que as imagens sejam entregues em perfeição. Sendo assim, há um tipo de comunicação lógica caracterizada pela despreocupação com a garantia de entrega dos dados, e também não há o firmamento de uma conexão entre os dispositivos da rede (apenas os dispositivos aptos a receber concluirão o recebimento do sinal de TV). Da mesma forma, numa transmissão ao vivo pela Internet, à 60Hz (para formar um segundo de vídeo é necessário ajuntar sessenta quadros), em que cada quadro é enviado em um único pacote, quando há uma perda de pacotes da imagem transmitida, logo em seguida haverá outro pacote de um quadro muito parecido em que mesmo com a perda do primeiro pacote não haverá perda na fluidez do vídeo.

2. **Jogos multiplayer on line**: Tal aplicação é um ótimo exemplo para caracterizar uma comunicação orientada para conexão com tratamento de erros para garantia de entrega de pacotes entre os adversários do jogo. A fim de melhor exemplificar, imagine um jogo da modalidade FPS (O. . . , 2019), que é executada localmente no computador de cada jogador, ou seja, os dados que essa aplicação disponibiliza na rede são referentes aos movimentos e golpes executados durante uma partida, os dados da posição do oponente são lidos da rede e posteriormente renderizados pela aplicação local. Então o mapeamento do jogo é realizado através da marcação de pontos em um gráfico de três dimensões e cada projétil disparado seja a representação de uma reta no mesmo gráfico. Assim, imagine que há apenas dois oponentes em uma mesma partida, **vermelho** e **azul**, em que ambos estão se movimentando pelo mapa. Para isso, ambos os computadores dos jogadores devem possuir uma conexão firme, uma vez que se a informação a cerca da posição do jogador **vermelho** não chegar corretamente ao computador do jogador **azul**, o jogador **azul** verá o personagem **vermelho** em uma posição diferente da correta, fazendo com que o jogo perca a sincronia. Dessa maneira vê-se a necessidade de uma comunicação persistente que garanta a troca de infomrações para o bom funcionamento de determinada aplicação.

Através desses dois exemplos, observa-se que há diversas possibilidades de implementações de tipos de comunicação, porém a comunicação lógica é escolhida de acordo com a necessidade da aplicação e de acordo com o levantamento da opção que melhor atenda. No exemplo da transmissão ao vivo, caso seja solicitado uma atualização que permita rever alguma parte do vídeo, seria necessário salvar os quadros de imagem recebidos, para isso, talvez a melhor opção seria garantir a persistência de toda informação transmitida, implementando um procedimento de correção de erros, mas talvez não seja necessário estabelecer uma conexão firme entre o transmissor e o receptor. Logo, é possível concluir que os casos variam de

acordo com a aplicação e dos desenvolvedores da aplicação.

Por fim, a comunicação lógica é caracterizada pela disponibilidade de recursos de transporte dos dados, não pela forma como os dados são transportados. A forma como os dados são transportados cabe à seção inferior da arquitetura **TCP/IP**.

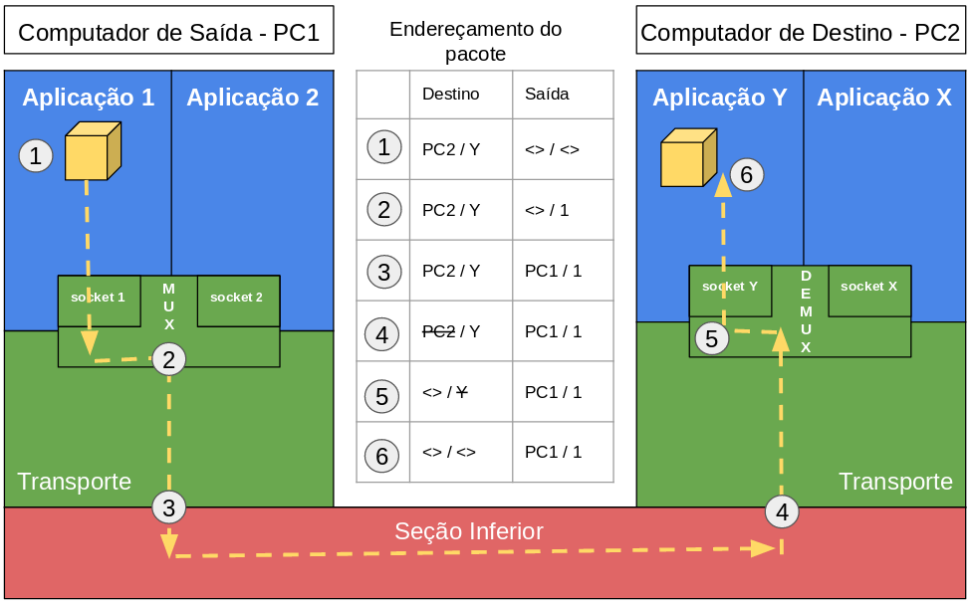


Figura 4 – Ecaminhamento e montagem do endereçamento através da multiplexação e demultiplexação.

2.3 MULTIPLEXAÇÃO E DEMULTIPLEXAÇÃO

Para um segundo momento, imagine que tanto a aplicação de transmissão ao vivo quanto o jogo FPS, da seção anterior, estão em execução no mesmo instante em um hospedeiro. Dado que, no geral, um computador possui apenas uma placa de rede, e por consequência apenas um endereço **IP**, há a necessidade e identificar a aplicação para a qual as informações recebidas na camada de rede sejam encaminhadas para a aplicação correta. Esse trabalho é, também, executado pela camada de transporte, uma vez que aplicação deve preocupar-se apenas com as informações de sua propriedade, e de sua hierarquia na camada de protocolos, cabendo a alguma camada inferior, tratar e redirecionar os dados corretamente para cada software desejado. Tal serviço é chamado de **Multiplexação** e **Demultiplexação** que é realizado de forma lógica pela configuração de portas virtuais para os *sockets*(escutadores) da rede.

Posteriormente, os principais protocolos que envolvem a camada de transporte serão detalhados e a maneira como ocorre a multiplexação e demultiplexação para cada um desses será melhor explicada. Entretanto, de maneira geral não há muita diferença entre esse serviço nos dois protocolos e pode-se explicar e visualizar como na Figura 4.

Algo que ainda não fora citado é o motivo pelo qual a arquitetura **TCP/IP** também é chamada de pilha de protocolos. Tal fato é explicado pelo motivo de que a comunicação inicia-se na camada de aplicação e à medida que é encaminhada para a parte física da rede (fios de fibra ópitca, fios de cobre, sinais de radio e etc.) há uma série de pacotes que são superpostos pelas camadas inferiores. Já no momento em que os dados chegam até o hospedeiro destinatário, os pacotes são "abertos"para a conclusão do trabalho do protocolo detentor do pacote, em seguida o pacote da camada superior é "retirado"iniciando o trabalho do protocolo da camada superior. Assim, a pilha de protocolos é uma estrutura que configura a ordem em que os pacotes são criados e destruídos à medida em que são acionados pelos recursos da rede, dessa forma, temos que em uma estrutura do tipo pilha, o topo é identificado pelo primeiro objeto inserido.

Para melhor explicar o funcionamento da multiplexação e da demultiplexação, imagine um ambiente de rede composto por um computador do tipo cliente que está em comunicação com dois servidores diferentes. Os servidores são identificados por **SVR-1** e **SVR-2**, os endereços **IP** de cada servidor são **IP-SVR-1** e **IP-SVR-2** respectivamente, além de que cada servidor é responsável por uma aplicação diferente: em **SVR-1** está em execução a aplicação **APP-A** e em **SVR-2** a aplicação **APP-B**. Já o computador cliente é identificado por **CLI** e seu endereço **IP** é **IP-CLI** em que ambas as aplicações **APP-A** e **APP-B** estão execução ao mesmo tempo. A tabela a seguir é mais instrutiva para visualização da distribuição da estrutura descrita:

Tabela 1 – Tabela de Hospedeiros.

Tipo de máquina	Nome	Endereço IP	Aplicação	ID da Aplicação
Servidor	SVR-1	IP-SVR-1	APP-A	APP-A-SVR-1
Servidor	SVR-2	IP-SVR-2	APP-B	APP-B-SVR-2
Cliente	CLI	IP-CLI	APP-A APP-B	APP-A-CLI APP-B-CLI

De posse da Tabela 1 utilizaremos uma linguem de mais alto nível quanto aos identificadores da camada de rede (endereço **IP**) e os identificadores da camada de transporte (número da porta virtual). Deixaremos especificação da numeração dos sockets e do endereço **IP** para outro momento.

2.3.1 Multiplexação

Tendo em vista que o computador **CLI** possui apenas um endereço **IP**, ou seja, apenas uma única via de comunicação, enquanto possui duas aplicações em execução, cada qual comunicando-se ao mesmo tempo com ambos os servidores **SVR-1** e **SVR-2**, é evidente a necessidade de um serviço que permita o **envio** de mensagens para os

dois servidores através de uma mesma via. Esse serviço é chamado de **Multiplexação** e consiste no escalonamento e na gerência do envio de dados através da camada de **Rede**.

Para isso, a camada de transporte necessita identificar cada aplicação da rede através de um número, além da identificação do **IP** do servidor e do **IP** do cliente. Dessa maneira, o pacote da camada de transporte conterá em seu cabeçalho os identificadores: Endereço **IP** do cliente, identificador da aplicação no cliente, endereço **IP** do servidor, identificador da aplicação no servidor; para que seja possível escalonar o envio de pacotes das aplicações e para que a **Demultiplexação** ocorra de forma correta. Uma forma fácil de ilustrar esse serviço é através do uso de um Multiplexador contido em circuitos integrados, como mostrado na imagem 5: As portas de entrada recebem os dados da aplicações, as portas de controle do **MUX** é o identificador da aplicação e a saída do circuito representa a seção inferior da arquitetura **TCP/IP**.

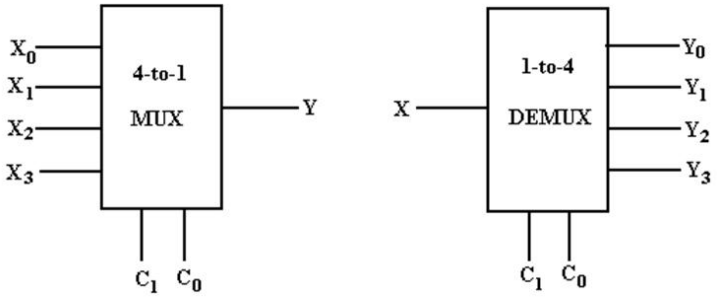


Figura 5 – Exemplo de multiplexador de quatro entradas e demultiplexador de quatro saídas.

Fonte: Disponível em: <http://2.oxmf.ab-renovation74.fr/block-diagram-of-4-to-1-multiplexer.html>. Acesso em: 07 de out. de 2019.

Portanto, para que uma aplicação consiga realizar sua comunicação, esta deve ser identificada no cliente e no servidor, logo temos que os identificadores de **APP-A** são **APP-A-CLI** e **APP-A-SVR-1** para o cliente e servidor respectivamente. O mesmo deve ser feito para **APP-B**. Logo, quando **APP-A** está enviando informações do cliente para o servidor, o cabeçalho do pacote de transporte conterá: **IP-CLI**, **APP-A-CLI**, **IP-SVR-1** e **APP-A-SVR-1** para que a demultiplexação ocorra de forma correta.

2.3.2 Demultiplexação

Ainda fazendo uso do cenário hipotético descrito pela Tabela 1 imagine que um pacote de **SVR-1** e outro de **SVR-2** são recebidos pelo computador **CLI**. Agora visto que há dois pacotes de aplicações diferentes, é necessário redirecioná-los para as aplicações corretas, então vê-se a necessidade do identificador da aplicação. Logo, vamos enumerar dois pacotes diferentes chegando até o computador **CLI**:

1. **IP-SVR-1**, **APP-A-SVR-1**, **IP-CLI**, **APP-A-CLI**
2. **IP-SVR-2**, **APP-B-SVR-2**, **IP-CLI**, **APP-B-CLI**
3. **IP-SVR-1**, **APP-B-SVR-2**, **IP-CLI**, **APP-B-CLI**

É nesse momento em que entra em ação o serviço de **Demultiplexação**, tratando de encaminhar o pacote **1** para a aplicação **APP-A** e o pacote **2** para a aplicação **APP-B**. Entretando, o pacote **3** não será reconhecido pelo demultiplexador, uma vez que está recebendo de **SVR-1** um pacote referente a uma aplicação com o identificador **APP-B-SVR-2**, sendo que é um pacote inesperado, portanto será desconsiderado e não chegará até a aplicação desejada.

2.4 UDP - *USER DATAGRAM PROTOCOL*

Após o entendimento genérico da camada de transporte, pode-se esclarecer o funcionamento dos dois principais protocolos dessa seção da arquitetura **TCP/IP**, que são responsáveis pela implementação do que hoje conhece-se como *Internet*. O primeiro, e mais básico, protocolo que abordaremos é o **UDP** uma vez que este é uma forma simples e direta de comunicação entre a camada de aplicação e a camada de rede, provendo serviços de forma rápida e econômica.

A simplicidade deste protocolo é justificada pelo modo como a comunicação lógica é estabelecida, dado que podemos descrever o **UDP** como um simples acréscimo ao protocolo ajacente **IP**. Uma vez que temos um protocolo não orientado para conexão, não há necessidade de estabelecer uma comunicação entre dois pontos; as aplicações que utilizam o **UDP** enviam seus dados sem necessidade de garantia na entrega dos pacotes, ou de correção de possíveis erros de transmissão. Logo esse método possui pontos positivos no quesito velocidade de execução, e por isso muitas aplicações escolhem implementá-lo quando há certo percentual de confiabilidade de entrega dos pacotes sem necessidade de correção em determinada rede.

Um bom motivo pelo qual as aplicações utilizam o **UDP** como protocolo de transporte, é a possibilidade de implementar parte da comunicação lógica dentro da própria aplicação, visto que esse protocolo é "vazio"quanto aos métodos lógicos de conexão. Então, determinada aplicação pode utilizar a não orientação para conexão para criar uma sub camada entre a aplicação e a camada de transporte a fim de implementar uma orientação de conexão como uma conexão criptografada, uma correção de erros específica de determinados dados, um método de sincronização no envio de pacotes e etc. Dessa forma, tal procolo é visto com positividade pela possibilidade de edição.

Retomando o exemplo **1** dado na seção **2** deste capítulo, em que desenvolveu-se uma aplicação de transmissão ao vivo de vídeo, o ***User Datagram Protocol*** seria uma ótima escolha para esse serviço. Dado que necessita-se de uma rápida transmissão, sem necessidade de garantir a entrega de todos os pacotes, uma vez que a rede possui confiabilidade razoável nesse quesito. Um exemplo real que poderia utilizar o **UDP** é o **Vo-IP** ou Voz por IP, que é um serviço de telefonia via Internet; assim como a transmissão ao vivo de vídeo, nessa aplicação, a perda de poucos pacotes não atrapalharão a conversação e o uso desse protocolo é benéfico quando à velocidade de entrega dos pacotes. Além disso, há apenas a necessidade de ordenar a chegada dos pacotes de voz, o que pode ser implementado de forma simples dentro da própria aplicação.

2.5 TCP - *TRANSMISSION CONTROL PROTOCOL*

Há uma vasta gama de métodos que dão nome a este protocolo e fazem jus à função da garantia e confiabilidade de entrega dos dados quando utiliza-se o **TCP**. Para detalhar como o Transmission Control Protocol é capaz de conduzir a informação de maneira rápida e confiável, utilizaremos nessa seção uma linguagem mais abstrata que se preocupa em relatar as funções ao invés de explicitar o *modus operandi* de cada propriedade do protocolo.

O **TCP** trata-se de um protocolo orientado para conexão, ou seja, durante um instante de tempo há uma conexão fixa entre dois hospedeiros da rede. É possível ilustrar essa função através do cenário de uma ligação telefônica, em que cada telefone se caracteriza como um hospedeiro: o telefone responsável por realizar a ligação é o remetente da comunicação e o destinatário é quem recebe a chamada. Antes do estabelecimento de uma conexão é necessário enviar uma mensagem de início da conexão, que no caso dos telefones é o toque do telefone destinatário. As mensagens trocadas são ilustradas como os dados de voz da ligação e a mensagem de encerramento da conexão é o corte do pulso da linha telefônica. Dessa maneira, o **TCP** estabelece uma conexão entre dois hospedeiros, porém de forma mais complexa através do salvamento de informações que serão utilizadas para a melhora na troca de informações de determinada rede.

2.5.0.1 Confiabilidade de transmissão

A melhora na comunicação entre dois hospedeiros na rede envolve dois processos principais: detectar possíveis bits errados e identificar quando algum bit de determinado pacote não chegou ao hospedeiro destinatário. Esses processos funcionam graças à adição de informações ao cabeçalho de um pacote **TCP** em que há

campos responsáveis por identificar a disposição dos bits na cadeia do datagrama e também há campos responsáveis pela contagem da quantidade de bits que o datagrama possui. Para isso há um recurso de resposta automática, que faz com que o destinatário emita ao remetente uma mensagem informando se o pacote chegou corretamente ou se houve algum erro no pacote o qual os bits de validação do cabeçalho **TCP** não conseguiram resolver, esse é o serviço chamado de **Realimentação do Destinatário** responsável por acionar o reenvio de um pacote que foi dado como corrompido no hospedeiro receptor.

Entretanto, o recurso de **Realimentação do Destinatário** também é acionado por temporizadores, a medida que, em determinado intervalo de tempo, o remetente não recebe resposta sobre o recebimento do pacote pelo destinatário. Isso faz com que esse serviço gere conflitos na sequencialidade dos dados recebidos; para ilustrar esse cenário basta imaginar que em uma rede sobrecarregada é natural que a transmissão de dados seja mais demorada, então é possível que um pacote **X** seja enviado numa primeira tentativa de sucesso através do datagrama $\mathbf{X}_1$, sendo que enquanto $\mathbf{X}_1$ "navega"pela rede o temporizador do protocolo **TCP** aciona o reenvio desse mesmo pacote **X**, através do datagrama $\mathbf{X}_2$, por inferir que $\mathbf{X}_1$ se "perdeu"no trajeto. Nesse cenário, é possível que ambos pacotes cheguem até o destinatário, logo é necessário tratar a inconsistência da chegada de um pacote repetido.

Lembrando que o serviço de implementação da comunicação lógica, no geral, é de responsabilidade da camada de transporte, o tratamento da chegada duplicada deve partir dessa mesma camada, e não da aplicação. Para tanto, há a adição de mais um campo no cabeçalho dos pacotes **TCP**, que enumera cada pacote, identificando-os sequencialmente, a fim de evitar que os dados duplicados cheguem à camada de aplicação. Após tantos tratamentos, assegura-se com maior confiabilidade a transmissão dos dados pela rede, porém, é notório que tais tratamentos de correção de bits em conjunto com a realimentação do destinatário causem certa demora na comunicação, por isso há um procedimento usado pelo protocolo **TCP** que visa diminuir o tempo de comunicação.

2.5.0.2 Transmissão Paralela

Imagine que um remetente tenta enviar três pacotes ($\mathbf{P}_1$, $\mathbf{P}_2$ e $\mathbf{P}_3$ em ordem, para determinado destinatário. É natural que, após a explicação a cerca dos tratamentos de integridade dos pacotes, imagina-se que o envio de um pacote $\mathbf{P}_{k+1}$ só se dá após o destinatário informar o recebimento correto do pacote $\mathbf{P}_k$. Porém, num cenário real, o tempo de tráfego entre duas máquinas é grande e esse tipo de envio seria impraticável por conta da demora na transmissão. Para isso, o **TCP** implementa um tipo

de envio "paralelo- apesar de que os dados são enviados sequencialmente, a noção de paralelismo deriva do fato de que envia-se um conjunto pacotes para se realizar o tratamento do recebimento, ao invés de se enviar um pacote por vez. Pela Figura 6, inspirada em (KUROSE; ROSS, 2012a), pode-se observar o fluxo de tratamento no envio de seis pacotes diferentes, os quais são separados em grupos de três elementos para o envio paralelo.

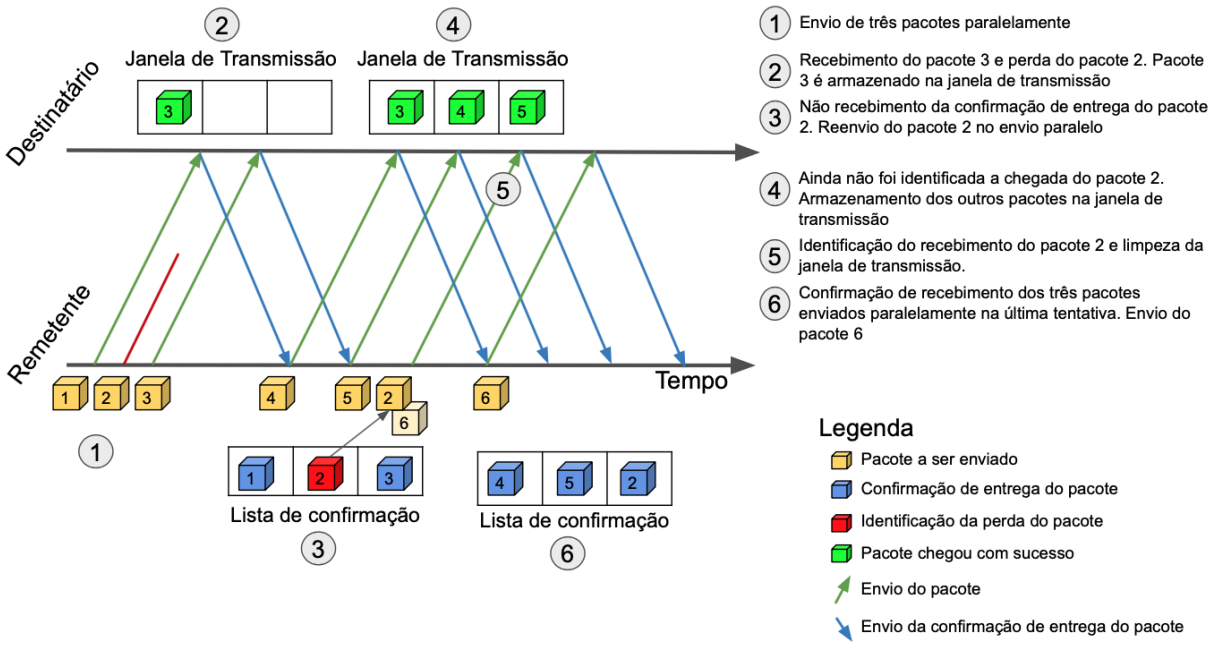


Figura 6 – Diagrama de comunicação paralela do TCP.

Antes de explicar os fatos da imagem, é necessário introduzir o conceito de janela de recepção e algumas notações utilizadas na imagem. A **Janela de Recepção** é um recurso implementado no hospedeiro destinatário sendo abordada como o armazenador de pacotes que permanecerão em *stand by* enquanto o protocolo realiza o trabalho de tratamento de recebimento dos pacotes anteriores. Ou seja, se um grupo de pacotes $\mathbf{P}_1$, $\mathbf{P}_2$ e $\mathbf{P}_3$ são enviados em conjunto, mas o pacote $\mathbf{P}_2$ não conseguiu chegar corretamente ao destinatário, o pacote $\mathbf{P}_2$ é guardado na **Janela de Recepção** enquanto aguarda a chegada de $\mathbf{P}_2$ para manutenção da sequencialidade a fim de entregar corretamente esses pacotes à aplicação.

2.5.0.3 Outros controles

O protocolo **TCP** possui muitos outros parâmetros que visam aumentar a eficiência da entrega dos datagramas, além de garantir a melhora na conexão. Tais controles são possíveis graças a troca de informações entre remetente e destinatários pelos cabeçalhos dos pacotes, além disso, o **TCP** é capaz de identificar problemas de transmissão causados na seção inferior da pilha **TCP/IP** e dar uma boa solução para os problemas dessas camadas.

Algumas das implementações que merecem destaque são:

1. **Controle de fluxo**: Reconhecimento da capacidade de armazenamento da **Janela de Recepção** no destinatário, para que os conjuntos de pacotes enviados pelo remetente não sobrecarreguem o armazenamento e haja um gasto desnecessário de transmissão de dados.
2. **Controle de Congestionamento**: É a função do protocolo responsável por reconhecer a latência e sobrecarga da rede e combater quedas de conexão e perda de pacotes por conta do mau funcionamento das camadas inferiores da arquitetura de rede. No geral, o **TCP** utiliza temporizadores nos cabeçalhos que reconhecem os tempos de transporte dos dados, além de que os multiplexadores e demultiplexadores desse protocolo são incrementados com *buffers* capazes de armazenar dados e tratar possíveis "gargalos"na chegada dos pacotes em determinado hospedeiro.

Por fim, é notável a quantidade de métodos e funções que o **TCP** implementa para garantir a confiabilidade na transmissão dos dados. Mais notável ainda, é saber que a **Web** é estruturada por este protocolo e observar que, mesmo com tantas validações, a velocidade de transmissão não é um problema. Apesar das grandes explicações, ainda há muito em que se aprofundar a cerca da camada de transporte, portanto, é válido indicar bons materiais como (KUROSE; ROSS, 2012b) e (TANENBAUM, 2002) caso deseja-se um maior aprofundamento no assunto.

3 COMUNICAÇÃO BÁSICA

Passada a parte teórica do funcionamento dos protocolos, dá-se início ao entendimento prático sobre a troca de informações entre *hosts*. Nesse sentido, necessita-se de uma estrutura capaz de proporcionar um ambiente que simule o comportamento da Internet, ou que esteja propriamente ligado à rede global. O **testbed FIBRE**, dará suporte para esse experimento, uma vez que é possível alocar recursos computacionais espalhados pelo Brasil, que estão ligados à Internet e que conseguem trocar informações entre si.

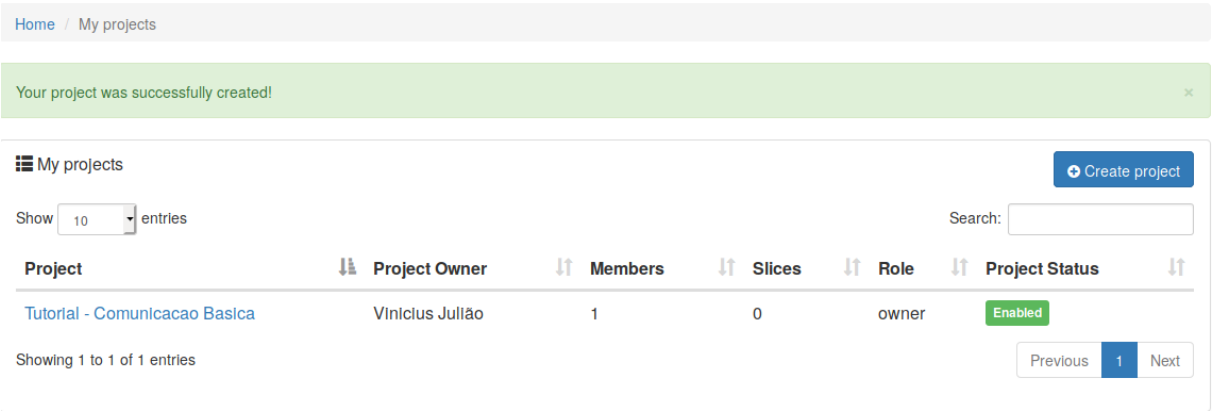
Para entender como fazer um primeiro uso da plataforma de acesso ao **testbed FIBRE** acesse o link: https://www.fibre.org.br/start-using-fibre/your-first-experiment/omf6-hello-world/. Nesse link há a alocação de apenas uma máquina virtual, entretanto como queremos uma comunicação entre dois *hosts* simulando o uso genérico dos protocolos, necessita-se alocar duas máquinas virtuais. As seções seguintes mostrarão como alocar duas máquinas e instanciá-las devidamente para gerar o ambiente do experimento.

Deste momento em diante, a demonstração será gerada a partir de imagens de recortes da tela do elaborador deste tutorial. Então atente-se aos detalhes mostrados, para que na tentativa de simular o que foi elaborado, obtenha-se o mesmo resultado.

3.1 CRIAÇÃO DO AMBIENTE

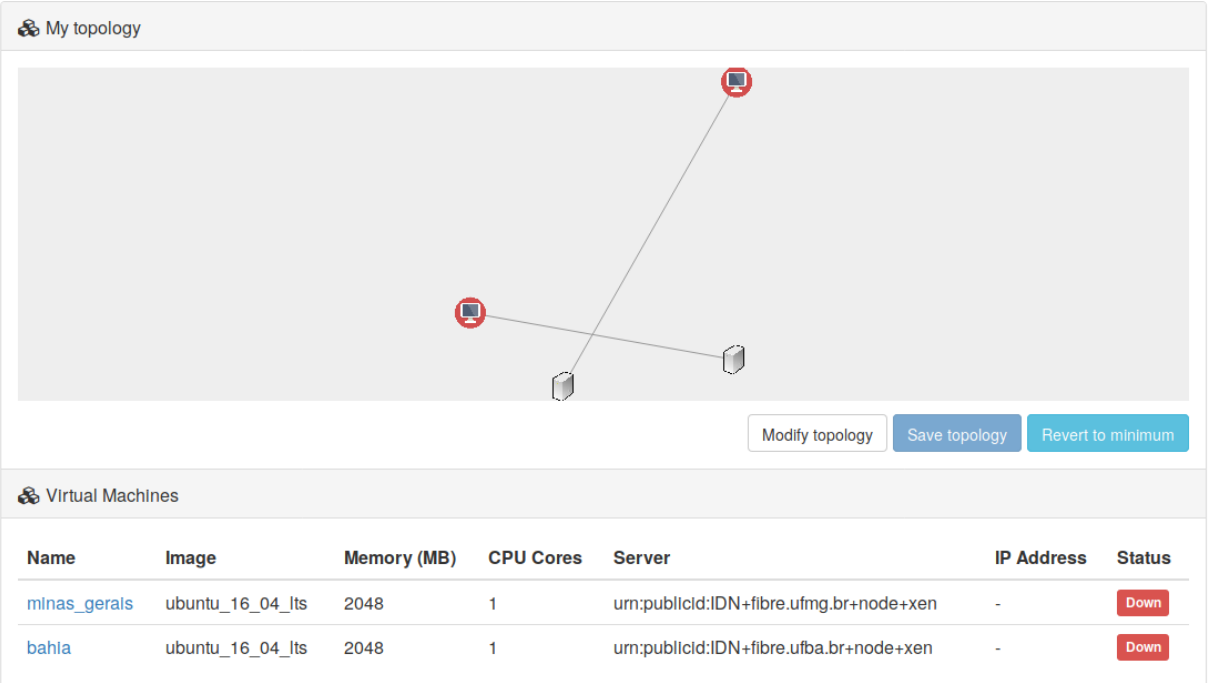
Para essa demonstração já há um projeto devidamente criado na seção "My projects"da plataforma **OMF6**, sendo assim selecionaremos tal projeto denominado "**Tutorial - Comunicacao Basica**".

Figura 7 – (Comunicação Básica) My projects



No projeto selecionado, criou-se um *slice* contendo duas máquinas virtuais (VM), uma delas de domínio fibre.ufba.br (Bahia) e outra de domínio fibre.ufmg.br (Minas Gerais). Assim, pode-se executar a troca de informações entre máquinas fisicamente muito distantes que farão uso da rede global para a troca tal comuniação.

Figura 8 – (Comunicação Básica) Detalhes do slice



Dada a seleção do *slice*, agora é o momento de realizar uma reserva no ambiente do testbed. Isso quer dizer que durante todo o tempo selecionado para a reserva, os recursos escolhidos no *slice* estarão disponível para uso. Após a criação da reserva é o momento de iniciar o experimento, assim como foi realizado no link de primeiro uso da plataforma **OMF6**.

Com o experimento em execução, as máquinas são endereçadas com um IP é preciso deixar salvo esses valores pois serão usados na comunicação entre os hosts. Após descobrir o endereço de cada máquina, estamos no passo de implementar o socket de comunicação de cada protocolo.

Figura 9 – (Comunicação Básica) Endereços IP das máquinas

Virtual Machines

Name	Image	Memory (MB)	CPU Cores	Server	IP Address	Status
minas_gerais	ubuntu_16_04_lts	2048	1	urn:publicid:IDN+fibre.ufmg.br+node+xen	10.142.13.128	UP and Ready
bahia	ubuntu_16_04_lts	2048	1	urn:publicid:IDN+fibre.ufba.br+node+xen	10.144.13.94	UP and Ready

Wireless Nodes

3.2 USANDO UDP

Para essa seção, é importante ressaltar que através da observação do comportamento da troca de dados de cada protocolo, poder-se-a comprovar as explicações realizadas na parte teórica deste tutorial. Portanto as imagens que seguem serão analisadas e brevemente explicadas, demonstrando o caráter da camada de aplicação.

Antes do início da descrição das implementações, vale ressaltar que a linguagem utilizada para o desenvolvimento dos programas aqui utilizados é [1], devido à facilidade de escrita e legibilidade que os códigos de tal linguagem possuem. Além disso, exemplos de utilização de sockets em **Python** são de fáceis de se encontrar em uma busca na web, inclusive providos pela própria comunidade que dá suporte a esta teconologia. Nesse mesmo cenário, os códigos implementados foram inspirados nas disponibilizações da página Socket Python Brasil[2].

Para a inspiração desse programa, imagine que determinado serviço disponível na internet necessite de reconhecer os nomes das pessoas que utilizam tal serviço. Nosso alfabeto é composto por letras minúsculas e maiúsculas, no entanto sabe-se que, apesar da lei gramatical a cerca de nomes serem escritos com o primeiro caractere maiúsculo, só há a necessidade de reconhecer se o nome de um usuário é igual ao de outro. Sendo assim, independente como a forma que o usuário digitou seu nome, se todas as letras em minúsculo ou maiúsculo, ou até mesmo havendo uma mescla, defini-se um padrão para comparar e salvar os nomes.

O padrão adotado pelo programa é o armazenamento de nomes com todas as letras em maiúsculo. Logo para esse programa, haverá um servidor capaz de colocar todas as letras de uma palavra em minúsculo e enquanto há o lado do cliente enviará as informações para o servidor que responderá logo em sequência.

Para esse experimento, vamos utilizar os códigos disponíveis no **Anexo A** e

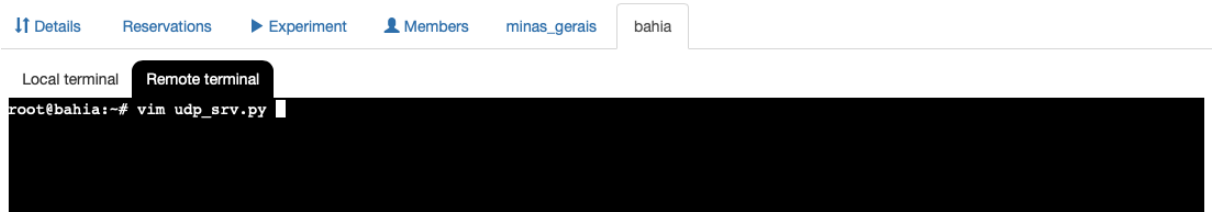
[1] https://www.python.org/
[2] https://wiki.python.org.br/SocketBasico

Anexo B, que representam o lado cliente e servidor respectivamente. Definiremos que a máquina **bahia** será o nosso servidor e a máquina **minas_gerais** o lado cliente. Então para isso, devemos saber qual é o **endereço IP** da máquina **bahia** que é o nosso servidor, para que o lado cliente consiga endereçar a conexão. Além disso, esse exercício convencionou que a porta identificadora do socket do servidor é de número **12000**.

Note ainda, que não houve necessidade de convencionar uma porta para o socket cliente, uma vez que o sistema operacional incumbe-se de realizar essa tarefa. Além disso, a porta de conexão com o servidor é de interesse geral, logo precisa-se de um identificador fixo. Já o cliente é apenas respondido por um servidor, logo o servidor responderá exatamente para a porta do cliente que enviou informação ao prestador do serviço.

Para realizar esse experimento, utilizaremos o editor de texto **Vim**[3] no qual copiaremos os códigos dos anexos e criaremos os arquivos necessários. Então, na máquina **bahia** execute:

Figura 10 – (Comunicação Básica) Editor de texto para servidor UDP



Coloque o código do **Anexo B** nesse aquivo e então salve-o.

[3] https://www.vim.org/

Figura 11 – (Comunicação Básica) Código para servidor UDP

Details Reservations Experiment Members minas_gerais bahia

Local terminal Remote terminal

```
from socket import *

SERVER_IP = ''
SERVER_PORT = 12000
ORIGIN = (SERVER_IP, SERVER_PORT)

serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(ORIGIN)
print('Servidor em execucao')
while 1:
        message, clientAddress = serverSocket.recvfrom(2048)
        message = message.decode('utf-8')
        print(clientAddress, message)
        modifiedMessage = message.upper() + ' - OK'
        serverSocket.sendto(modifiedMessage.encode(), clientAddress)
~
~
~
~
~
~
~
~
~
~
~
:x
```

Repita o passo anterior na máquina **minas_geais**, atentando-se para o nome do arquivo e o código copiado deve ser o mesmo do **Anexo A**.

Figura 12 – (Comunicação Básica) Editor de texto para cliente UDP

Details Reservations Experiment Members minas_gerais bahia

Local terminal Remote terminal

```
root@minas-gerais:~# vim udp_cli.py
```

Figura 13 – (Comunicação Básica) Código para cliente UDP

Details | Reservations | Experiment | Members | minas_gerais | bahia

Local terminal | Remote terminal

```
from socket import *
import time

SERVER_IP = '10.144.13.94' #IP do servidor
SERVER_PORT = 12000 #porta udp do servidor
SERVER = (SERVER_IP, SERVER_PORT)
udp_client_socked = socket(AF_INET, SOCK_DGRAM)

print('Socket cliente em execucao\nPara sair use CTRL+X')
print('=== Digite caracteres minusculos: ')
send_msg = input()
received_msg = None
while send_msg != '\x18':
    udp_client_socked.sendto(send_msg.encode(), SERVER)
    time.sleep(0.5) #delay proposital para sequencialidade de comunicação
    received_msg, _ = udp_client_socked.recvfrom(2048)
    print('=== Resposta do servidor: \n', received_msg.decode('utf-8'), '\n')
    print('=== Digite caracteres minusculos: ')
    send_msg = input()

udp_client_socked.close()
~
~
~
~
~
:x
```

Primeiramente, execute o código no cliente, e digite algumas palavras. Note que não haverá resposta recebida. Então pressione Ctrl+C para abandonar a execução do cliente.

Figura 14 – (Comunicação Básica) Comando de execução do cliente UDP com servidor offline

Details | Reservations | Experiment | Members | minas_gerais | bahia

Local terminal | Remote terminal

```
root@minas-gerais:~# python3 udp_cli.py
```

Figura 15 – (Comunicação Básica) Executando cliente UDP com servidor offline

Details | Reservations | Experiment | Members | minas_gerais | bahia

Local terminal | Remote terminal

```
root@minas-gerais:~# python3 udp_cli.py
Socket cliente em execucao
Para sair use CTRL+X
=== Digite caracteres minusculos:
abcd
teste
testando
blablabla
^CTraceback (most recent call last):
  File "udp_cli.py", line 16, in <module>
    received_msg, _ = udp_client_socked.recvfrom(2048)
KeyboardInterrupt
root@minas-gerais:~#
```

Execute o lado servidor (bahia):

Figura 16 – (Comunicação Básica) Comando de execução do servidor UDP

Details | Reservations | Experiment | Members | minas_gerais | bahia

Local terminal | Remote terminal

```
root@bahia:~# python3 udp_srv.py
Servidor em execucao
```

Execute o lado cliente novamente, digite algumas palavras e observe as saídas do lado cliente e do lado servidor:

Figura 17 – (Comunicação Básica) Execução do cliente UDP em funcionamento

Details | Reservations | Experiment | Members | minas_gerais | bahia

Local terminal | Remote terminal

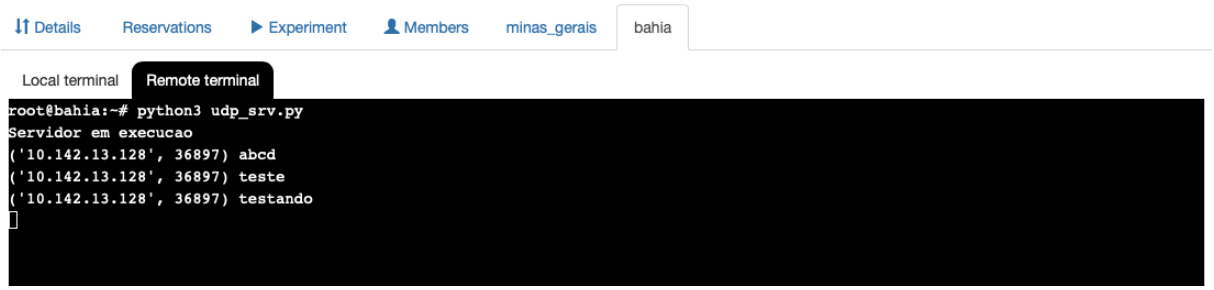
```
root@minas-gerais:~# python3 udp_cli.py
Socket cliente em execucao
Para sair use CTRL+X
=== Digite caracteres minusculos:
abcd
=== Resposta do servidor:
 ABCD - OK

=== Digite caracteres minusculos:
teste
=== Resposta do servidor:
 TESTE - OK

=== Digite caracteres minusculos:
testando
=== Resposta do servidor:
 TESTANDO - OK

=== Digite caracteres minusculos:
```

Figura 18 – (Comunicação Básica) Execução do servidor UDP em funcionamento



Por fim, foi possível reconhecer a simplicidade no uso do protocolo **UDP** que por sua vez realiza apenas um simples envio de mensagem. Tente excluir as linhas 15, 16 e 17 que estão no **Anexo A** e observe que as mensagens serão enviadas ao servidor e o servidor responderá mesmo que não haja o recebimento da mensagem pelo lado cliente. Essa etapa fica para o leitor executar e observar o comportamento.

3.3 USANDO TCP

A mesma abordagem feita para o experimento que utiliza **UDP** será aplicada nessa seção, mostrando o comportamento do protocolo de acordo com a execução do experimento no **testbed FIBRE**. Então para essa seção será utilizada a mesma estrutura das máquinas **bahia** e **minas_gerais** que é o slice criado com duas máquinas.

Para utilizar o protocolo **TCP** faremos apenas uma observação e análise das mensagens recebidas pelo lado servido, ao invés de recomunicar com o lado cliente. Os códigos que serão implementados para isso, encontram-se no **Anexo C** e **Anexo D**.

Lembre-se que há uma de código responsável por identificar o endereço **IP** do servidor. Logo é necessário alterá-la de acordo com o endereço da máquina que executará o provedor do serviço, que no nosso caso será a máquina **minas_gerais**, e o cliente será a máquina **bahia**. Observando a imagem seguinte mostraremos como criar os arquivos de código para o servidor através do editor **Vim**:

Figura 19 – (Comunicação Básica) Comando vim para servidor TCP

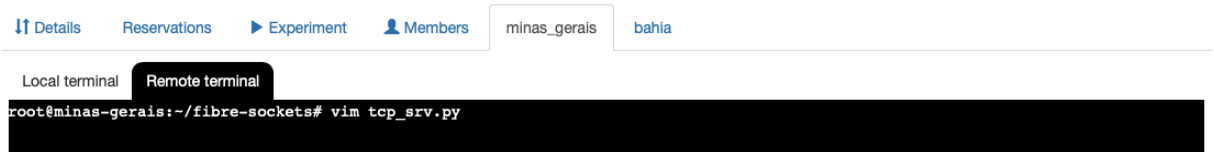


Figura 20 – (Comunicação Básica) Código para servidor TCP

Details Reservations Experiment Members minas_gerais bahia

Local terminal Remote terminal

```
import socket
HOST = ''
PORT = 5000
tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
orig = (HOST, PORT)
tcp.bind(orig)
tcp.listen(2)
while True:
    con, cliente = tcp.accept()
    print ('Concetado por', cliente)
    while True:
        msg = con.recv(1024)
        if not msg: break
        print (cliente, msg.decode('utf-8'))
    print ('Finalizando conexao do cliente', cliente)
con.close()
~
~
~
~
~
~
~
~
~
~
:x
```

Agora, na máquina **bahia** criaremos a parte cliente com o seguinte. Note que o cliente deve saber qual o endereço **IP** e a porta do socket do servidor:

Figura 21 – (Comunicação Básica) Comando vim para cliente TCP

Details Reservations Experiment Members minas_gerais bahia

Local terminal Remote terminal

```
root@bahia:~# vim tcp_cli.py
```

Figura 22 – (Comunicação Básica) Código para cliente TCP

Vamos tentar executar primeiramente a parte cliente antes de executar o servidor, tentando manter uma conexão e observando que será impossível dado que não há uma máquina disponível para receber a mensagem. O mesmo acontecerá se houver um servidor identificado pela porta **X** em execução, e tentar-se conectar em um servidor com uma porta de ID diferente de **X**. Tente realizar esse experimento alterando linha de código do **Anexo C** que identifica a porta do socket no servidor.

Observe que há uma mensagem de erro que identifica a incapacidade em estabelecer a conexão. Isso significa que esgotou-se o tempo para que a mensagem chegasse até o destino e fosse confirmado o recebimento dessa mensagem (lembre-se que o **TCP** é um protocolo com tratamento de comunicação, então a mensagem precisa ir e voltar com a confirmação de recebimento).

Figura 23 – (Comunicação Básica) Execução do cliente TCP com servidor offline

Details Reservations Experiment Members minas_gerais bahia

Local terminal Remote terminal

```
root@bahia:~# python3 tcp_cli.py
Traceback (most recent call last):
  File "tcp_cli.py", line 6, in <module>
    tcp.connect(dest)
ConnectionRefusedError: [Errno 111] Connection refused
root@bahia:~#
```

Inicie a execução do servidor:

Figura 24 – (Comunicação Básica) Execução do servidor TCP

Details Reservations Experiment Members minas_gerais bahia

Local terminal Remote terminal

```
root@minas-gerais:~# python3 tcp_srv.py
```

Reinicie a execução do lado cliente e digite algumas palavras.

Figura 25 – (Comunicação Básica) Execução do cliente TCP

Details Reservations Experiment Members minas_gerais bahia

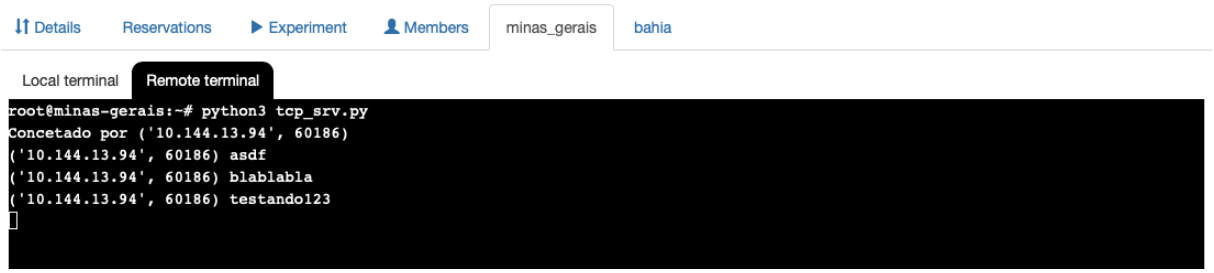
Local terminal Remote terminal

```
root@bahia:~# python3 tcp_cli.py
Para sair use CTRL+X

asdf
blablabla
testando123
```

Observe que a informação mostrada no servidor junto do **endereço IP** do cliente:

Figura 26 – (Comunicação Básica) Resultado da comunicação entre cliente e servidor TCP



Note que o resultado é muito parecido com o observado no uso do **TCP**, entretanto para desafiar o leitor a observar mais a fundo o a diferença entre o **TCP** e **UDP** sugere-se criar um novo experimento na plataforma do **testbedd FIBRE** em que há um *slice* contendo três máquinas, siga os seguintes passos fazendo uso dos códigos **UDP** e depois repita todo o proceso usando **TCP**:

1. Escolha uma máquina para ser o servidor e identifique o **endereço IP** dessa máquina.
2. Nos código do cliente, altere o **endereço IP** para valor obtido no passo anterior.
3. Execute o cliente nas duas outras máquinas restantes.
4. Digite alguns caracteres nos clientes e observe se os dados chegarão corretamente ao lado servidor.
5. Caso os dados de alguma das máquinas cliente não estejam chegando até o servidor, pare a execução da máquina que está comunicando corretamente.
6. Procure entender o motivo pelo qual o comportamento foi diferente.

3.4 QUESTÕES

1. Os códigos que implementam os experimentos feitos no testbed seriam categorizados na camada de transporte ou de aplicação?
2. É possível que haja dois sockets (um TCP e outro UDP) em execução ao mesmo tempo, cada qual com o memso número identificador? Faça o teste utilizando o código servidor do TCP e do UDP para responder essa pergunta.

4 SISTEMA DE MONITORAMENTO DE RECURSOS COMPUTACIONAIS

A internet e as grandes redes permitiram que, cada vez mais, os trabalhos fossem realizados de forma remota. No mundo dos recursos computacionais, a realização de tarefas à distância é o facilitador que permite o controle e o acesso à maquinas que estejam fisicamente distantes de seus gerentes. Atualmente, existem várias aplicações que permitem acesso remoto a computadores e também aplicações que permitem a visualização da disponibilidade de recursos de determinada máquina. Dois grandes exemplos, que são utilizados pela plataforma do **testbed FIBRE** é o **SSH** e o **Zabbix**, o **SSH** é um controlador de acesso remoto que através de um terminal bash permite que se controle outra máquina. Já o Zabbix é um monitor de recursos computacionais que dispõem a um escutador um relatório sobre o estado atual de um máquina, como memória, uso da CPU, espaço em disco, latência da rede e etc.

A título de curiosidade: o **SSH** é utilizado na janela em que de digita os comandos bash para a máquina alocada executar. Já o **Zabbix** é utilizado para monitorar os recursos de todos os nós disponíveis para uso no **testbed** e seleção daqueles que estão diponíveis, assim como mostra o link http://mon.fibre.org.br/zabbix.php?action=dashboard.view.

Para essa seção faremos uma implementação de um monitor de recurso de uma máquina, seguindo o passo a passo da implementação do código e por fim tomando uma decisão sobre qual protocolo utilizar - TCP ou UDP - para comunicar ao escutador o estado da máquina.

4.1 IMPLEMENTAÇÃO DO CÓDIGO

O objetivo desse tutorial não é praticar programação junto ao leitor, entretanto um passo a passo pode ser seguido para demonstrar as decisões de implementação tomadas no momento em que se desenvolveu a solução para um monitor de recursos computacionais.

Inicialmente, como na etapa da comunicação básica, aqui também será utilizado a linguagem **Python** pela facilidade de implementação do código e pela existência de bibliotecas que incluem métodos capazes de obter os recursos computacionais como se deseja. Para isso necessita-se da biblioteca **psutil** que será devidamente baixada à medida que necessita-se de executar o código.

Antes de tudo, deve-se planejar qual as informações que deseja-se obter do

servidor e para generalizar decidiu-se obter apenas o percentual de memória utilizada e o percentual de uso da CPU. Assim, dois comandos da biblioteca **psutil** serão de grande valia:

psutil.virtual_memory().percent: Obtém o percentual de memória utilizada na máquina.

psutil.cpu_percent(): Obtém o percentual do uso da CPU

Estipulamos que a cada três segundos, o servidor enviará uma informação ao cliente conectante informando esses dois dados. Para isso o comando **time.sleep(3)** servirá para realizar tal pausa. E por fim, escolheu-se a implementação via **TCP** para garantir que a informação a cerca da máquina chegue corretamente ao escutador, uma vez que serviços de monitoramento de recursos devem ser precisos e erros de comunicação devem ser reportados pois a qualidade da rede é uma boa métrica de outra variável a cerca da máquina.

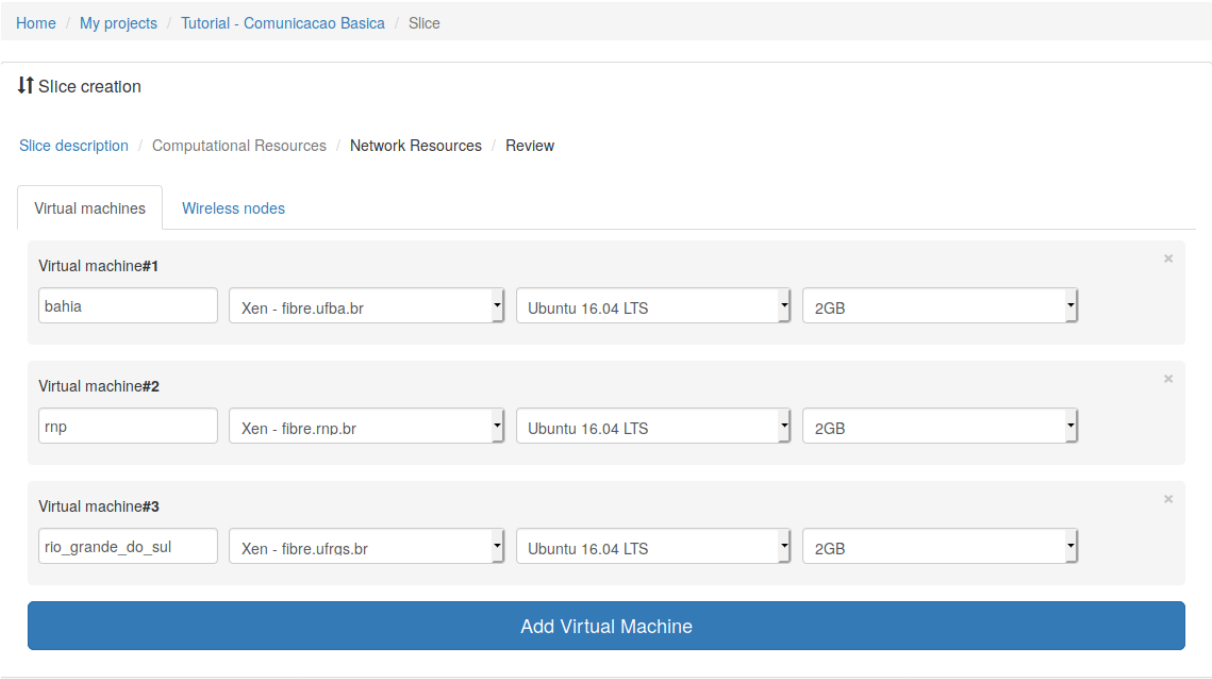
Até então, apenas o lado servidor foi implementado, e o tempo de espera de três segundos está implementada no servido. A razão por essa decisão é fácil de se imaginar; espera-se que uma máquina que presta um serviço de rede esteja executando outros muitos processos, e por isso caso a aplicação de monitoramento seja executada sem o devido cuidado, essa pode ser motivo de lentidão caso ocorra uma quantidade muito grande de requisições ao estado da máquina. Logo, é possível que o *delay* entre duas respostas do servidor seja maior ou igual a três segundos, desde que a parte cliente implemente, também, um certo atraso proposital para a comunicação.

A parte cliente, por sua vez, exerce um papel básico: solicitar o envio de informações da máquina servidor. Essa função foi implementada através do envio (por parte do cliente) de uma mensagem com um caractere espaço e após tal envio o aguardo ao recebimento de uma mensagem de resposta do servidor, que será exibida na tela. Os códigos devidamente implementados do lado cliente e do lado servidor podem ser encontrados nos anexos **E** e **F**, respectivamente.

4.2 EXECUÇÃO

Nesse experimento necessita-se criar um *slice* contendo três diferentes nós. Para nosso experimento instanciaremos as seguintes máquinas:

Figura 27 – (Sistema de Monitoramento) Criação do slice



Suponhamos que há duas máquinas das quais você (o leitor) é responsável por monitorar e ambas têm acesso à internet. Agora imagine que de tempos em tempos, os programas executados nessas máquinas dão problemas de *leaks* de memória ou então sobrecarga de execução e é seu papel intervir para sanar os problemas. Logo, para que você não precise a todo tempo esperar que as máquinas se sobrecarreguem, você implementa uma pequena aplicação que mostra o estado da memória e do uso da CPU, assim como mostrado no **Anexo E** e **Anexo F**.

Temos então que nesta simulação os dois servidores serão representados pelas máquinas **rio_grande_do_sul** e **bahia**, já o cliente será representado pela máquina **rnp**. Sendo assim, da mesma maneira como executado na seçao **Comunicação básica**, vamos implementar o código do servidor na parte servidor e o código cliente na parte cliente do ambiente:

Figura 28 – (Sistema de Monitoramento) Chamada do vim para criação do arquivo do Anexo E

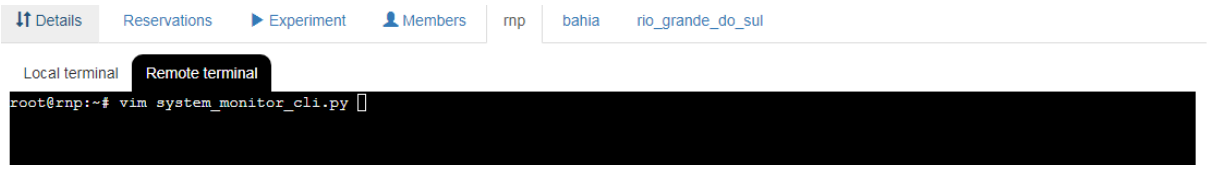


Figura 29 – (Sistema de Monitoramento) Colagem do Anexo E no arquivo do cliente rnp

Details | Reservations | Experiment | Members | rnp | bahia | rio_grande_do_sul

Local terminal | Remote terminal

```
import socket
import sys
HOST = sys.argv[1] #IP do servidor obtido por arugmento de execucao
PORT = 5000            # Porta que o Servidor esta
tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
dest = (HOST, PORT)
tcp.connect(dest)
while True:
    tcp.send ('__'.encode())
    modifiedSentence = tcp.recv(1024)
    print(modifiedSentence.decode('utf-8'))
tcp.close()
~
~
~
~
~
~
~
~
~
~
~
~
~
"system_monitor_cli.py" [noeol] 12L, 371C                    1,1           All
```

Figura 30 – (Sistema de Monitoramento) Chamada do vim para criação do arquivo do Anexo F na máquina rio_grande_do_sul

Details | Reservations | Experiment | Members | rnp | bahia | rio_grande_do_sul

Local terminal | Remote terminal

```
root@rio-grande-do-sul:~# vim system_monitor_srv.py
```

Figura 31 – (Sistema de Monitoramento) Colagem do Anexo F no arquivo do servidor rio_grande_do_sul

Details | Reservations | Experiment | Members | rnp | bahia | rio_grande_do_sul

Local terminal | Remote terminal

```
import psutil
import socket
import time

HOST = ''               # Endereco IP do Servidor
PORT = 5000             # Porta que o Servidor esta
tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
orig = (HOST, PORT)
tcp.bind(orig)
tcp.listen(1)
while True:
    con, cliente = tcp.accept()
    print ('Concetado por', cliente)
    while True:
        msg = con.recv(1024)
        if not msg: break
        print ("Cliente conectado",cliente)
        time.sleep(3)
        send_msg = 'cpu percent: '+str(psutil.cpu_percent())+'\
            - memory usage percent: '+ str(psutil.virtual_memory().percent)
        con.send(send_msg.encode())
    print ('Finalizando conexao do cliente', cliente)
    con.close()
~
~
~
"system_monitor_srv.py" [noeol] 23L, 709C                    1,1           All
```

Figura 32 – (Sistema de Monitoramento) Chamada do vim para criação do arquivo do Anexo F na máquina bahia

Details | Reservations | Experiment | Members | rnp | bahia | rio_grande_do_sul

Local terminal | Remote terminal

```
root@bahia:~# vim system_monitor_srv.py
```

Figura 33 – (Sistema de Monitoramento) Colagem do Anexo F no arquivo do servidor bahia

Details | Reservations | Experiment | Members | rnp | bahia | rio_grande_do_sul

Local terminal | Remote terminal

```
import psutil
import socket
import time

HOST = ''               # Endereco IP do Servidor
PORT = 5000             # Porta que o Servidor esta
tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
orig = (HOST, PORT)
tcp.bind(orig)
tcp.listen(1)
while True:
    con, cliente = tcp.accept()
    print ('Concetado por', cliente)
    while True:
        msg = con.recv(1024)
        if not msg: break
        print ("Cliente conectado",cliente)
        time.sleep(3)
        send_msg = 'cpu percent: '+str(psutil.cpu_percent())+'\
            - memory usage percent: '+ str(psutil.virtual_memory().percent)
        con.send(send_msg.encode())
    print ('Finalizando conexao do cliente', cliente)
    con.close()
~
~
~
"system_monitor_srv.py" [noeol] 23L, 709C                    1,1          All
```

Antes de inicializar a execução dos servidores, deve-se instalar a biblioteca da linguagem **Python** capaz de obter os dados dos recursos computacionais. Para isso é necessário instalar o instalador de bibliotecas do **Python 3**, o **pip3**. Para garantir que não haja problema no download do instalador de bibliotecas do **Pyhton 3**, execute o comando:

```
$ apt update
```

Então repita os passos seguintes:

Figura 34 – (Sistema de Monitoramento) Instalação do pip3 em rio_grande_do_sul

Details | Reservations | Experiment | Members | rnp | bahia | rio_grande_do_sul

Local terminal | Remote terminal

```
root@rio-grande-do-sul:~# apt install python3-pip
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libpython3-dev libpython3.5 libpython3.5-dev libpython3.5-minimal libpython3.5-stdlib
  python3-dev python3-setuptools python3-wheel python3.5 python3.5-dev python3.5-minimal
Suggested packages:
  python-setuptools-doc python3.5-venv python3.5-doc binfmt-support
The following NEW packages will be installed:
  libpython3-dev libpython3.5-dev python3-dev python3-pip python3-setuptools
  python3-wheel python3.5-dev
The following packages will be upgraded:
  libpython3.5 libpython3.5-minimal libpython3.5-stdlib python3.5 python3.5-minimal
5 upgraded, 7 newly installed, 0 to remove and 168 not upgraded.
122 not fully installed or removed.
Need to get 43.5 MB/43.8 MB of archives.
After this operation, 55.2 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Figura 35 – (Sistema de Monitoramento) Instalação da biblioteca psutil pelo pip3 em rio_grande_do_sul

Details | Reservations | Experiment | Members | rnp | bahia | rio_grande_do_sul

Local terminal | Remote terminal

```
root@rio-grande-do-sul:~# pip3 install psutil
```

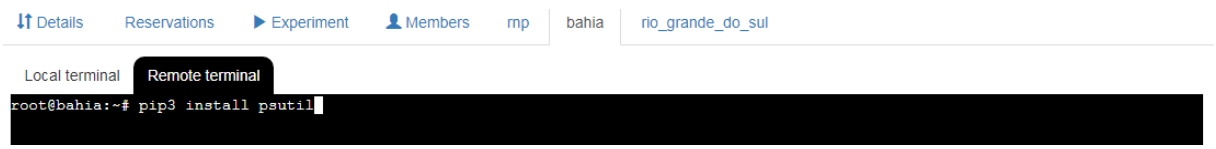
Figura 36 – (Sistema de Monitoramento) Instalação do pip3 em bahia

Details | Reservations | Experiment | Members | rnp | bahia | rio_grande_do_sul

Local terminal | Remote terminal

```
root@bahia:~# apt install python3-pip
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libpython3-dev libpython3.5-dev python3-dev python3-setuptools python3-wheel
  python3.5-dev
Suggested packages:
  python-setuptools-doc
The following NEW packages will be installed:
  libpython3-dev libpython3.5-dev python3-dev python3-pip python3-setuptools
  python3-wheel python3.5-dev
0 upgraded, 7 newly installed, 0 to remove and 150 not upgraded.
123 not fully installed or removed.
Need to get 38.0 MB of archives.
After this operation, 55.2 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

Figura 37 – (Sistema de Monitoramento) Instalação da biblioteca psutil pelo pip3 em bahia



Agora, vamos iniciar execução dos servidores:

Figura 38 – (Sistema de Monitoramento) Chamada da execução do servidor rio_grande_do_sul

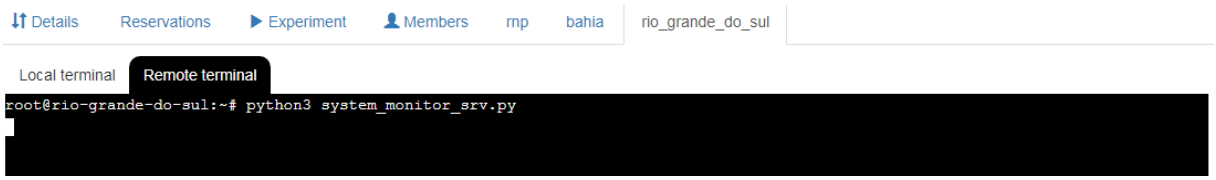
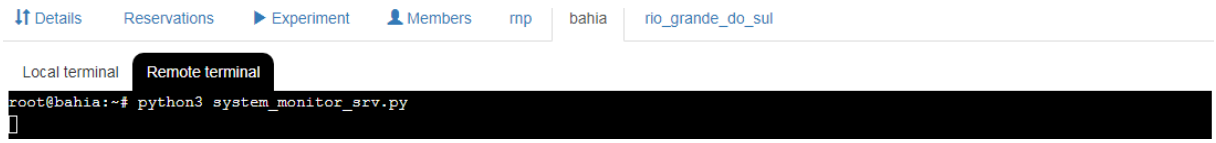
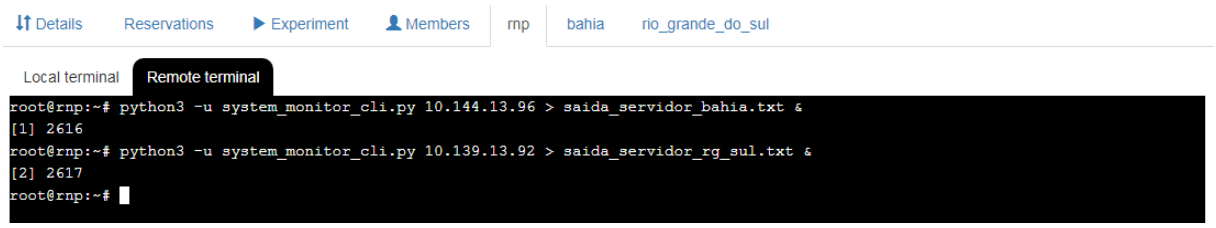


Figura 39 – (Sistema de Monitoramento) Chamada da execução do servidor bahia



Nesse ponto, temos apenas uma máquina cliente e duas máquinas servidor, então como um socket está conectado diretamente a outro socket, devemos executar duas instâncias do lado cliente. Para isso, dado que tem-se apenas uma janela **SSH**, vamos encaminhar as mensagens recebidas pelo cliente para um arquivo e executar os programas em segundo plano. Além do mais, é necessário saber quais os **endereços IP** das máquinas **bahia** e **rio_grande_do_sul** para que seja possível redirecionar a conexão de forma correta. Para isso, a implementação do lado cliente de execução está um pouco mais elaborada a fim de receber um parâmetro que corresponde ao **endereço IP** do servidor que deseja-se comunicar.

Figura 40 – (Sistema de Monitoramento) Chamada da execução do cliente para comunicar com cada servidor em segundo plano



É necessário saber qual é o ID do processo que cada chamada retornou para que seja possível encerrá-lo devidamente. O identificador do processo é o número que foi impresso assim que executou-se o programa em segundo plano, então guarde tal número para finalizar a execução do cliente. Para ler o que está sendo impresso nos arquivos basta executar o comando **cat** que o conteúdo atual será exibido:

Figura 41 – (Sistema de Monitoramento) Leitura do arquivo de saída da comunicação com rio_grande_do_sul

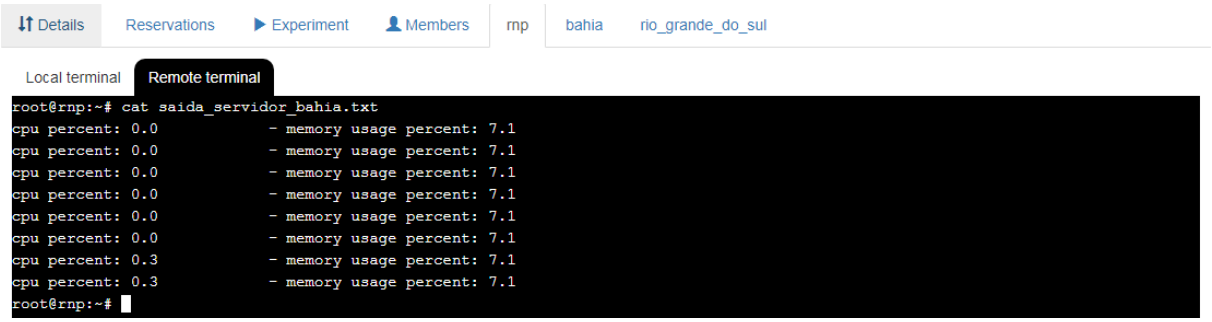
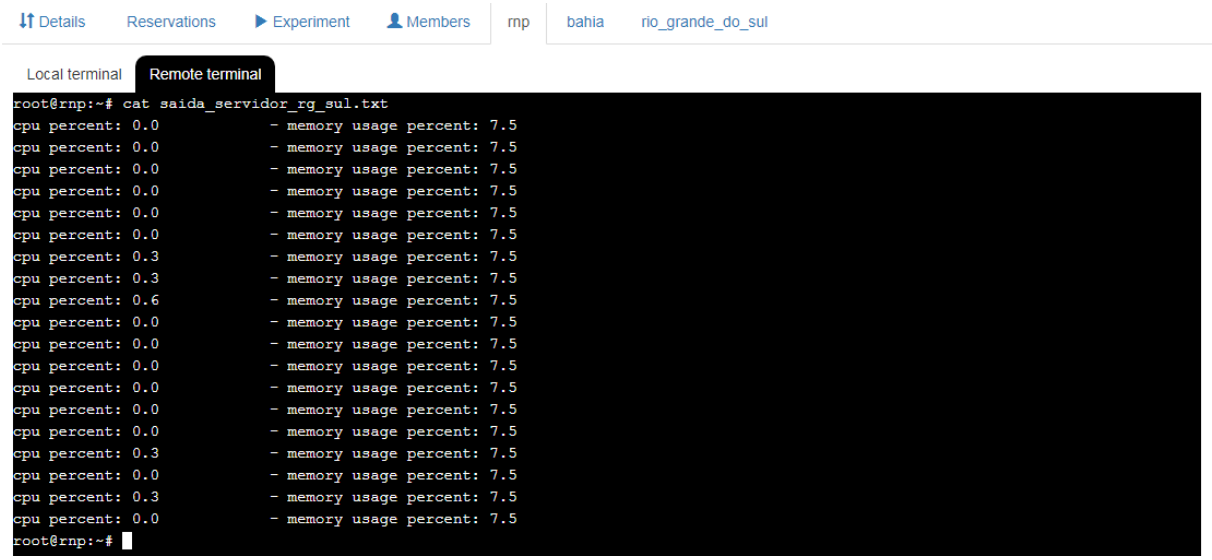


Figura 42 – (Sistema de Monitoramento) Leitura do arquivo de saída da comunicação com bahia



Visualizando a saída exibida nas máquinas **rio_grande_do_sul** e **bahia** observa-se que há o dado da conexão de um mesmo **endereço IP**, porém um numero diferente de porta de conexão. Isso é explicado pelo fato de que ambos os servidores estão comunicando com a mesma máquina, entretanto para os protocolos da camada de transporte, a aplicação que contém o *socket* de número **X** é totalmente diferente da aplicação que tem *socket* de número **Y** e portanto. Essa diferenciação no número identificado dos *sockets* ocorreu no momento em que, no lado cliente, criou-se duas

instâncias diferentes do programa, uma para cada servidor. Assim, haverá o processo de multiplexação e demultiplexação dos pacotes trocados entre os servdores e o cliente.

Figura 43 – (Sistema de Monitoramento) Exibição da saída da comunicação entre cliente e rio_grande_do_sul

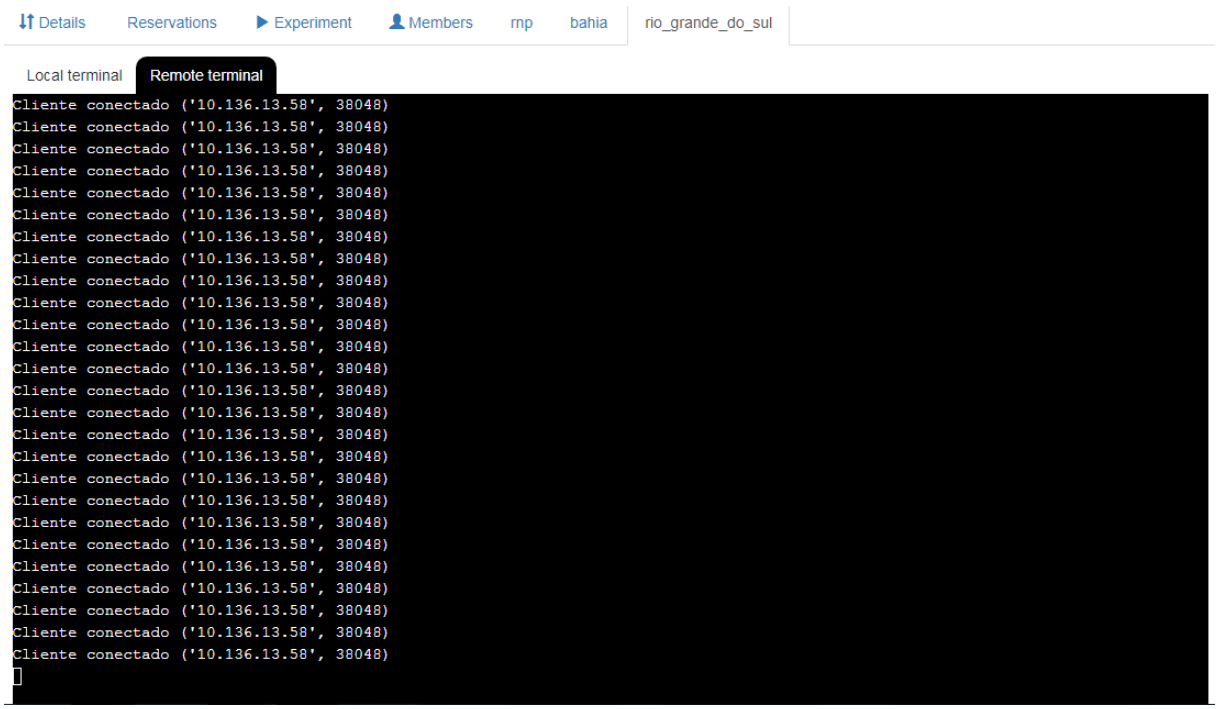
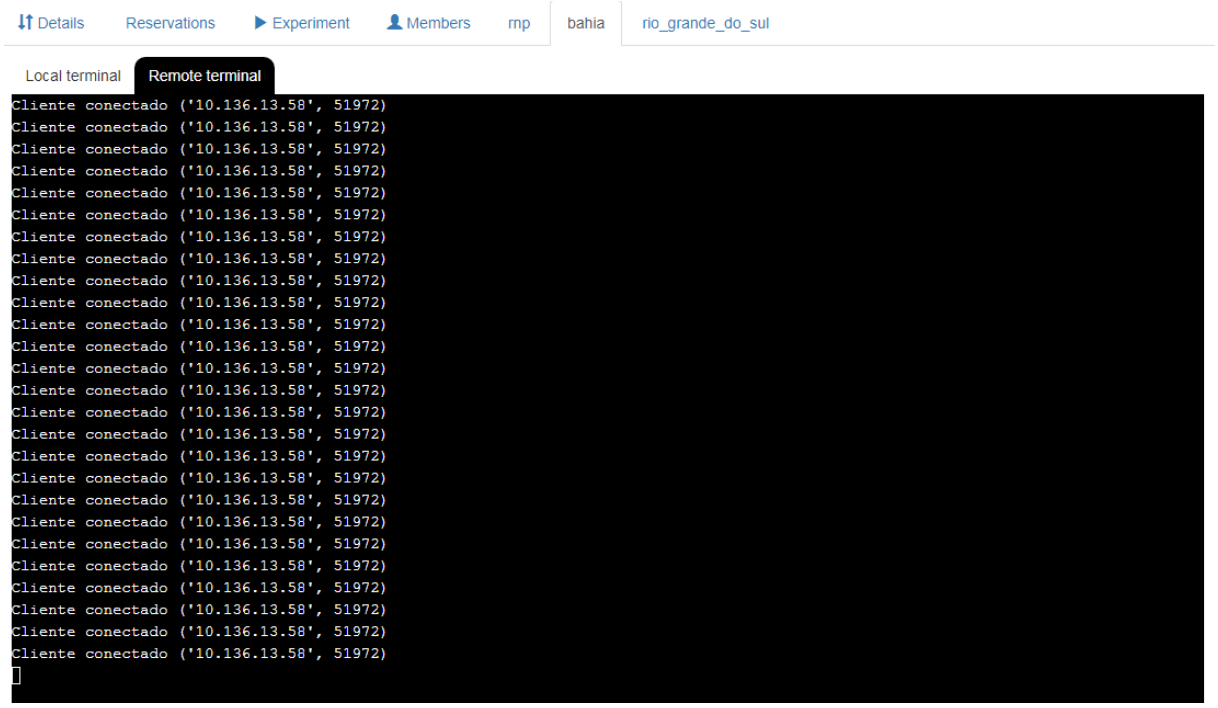


Figura 44 – (Sistema de Monitoramento) Exibição da saída da comunicação entre cliente e bahia



Após a observação e análise do experimento, encerre a execução dos programas cliente através do comando **kill**, e do identificador do processo que fora salvo na chamada de execução em segundo plano. Note que ao terminar a execução de algum cliente, logo em seguida o servidor que esteve conectado com tal host identifica o fim da conexão.

Figura 45 – (Sistema de Monitoramento) Finalizando execução dos clientes que estão em segundo plano de acordo com o número do processo

Figura 46 – (Sistema de Monitoramento) Resposta do servidor rio_grande_do_sul ao encerrar o cliente

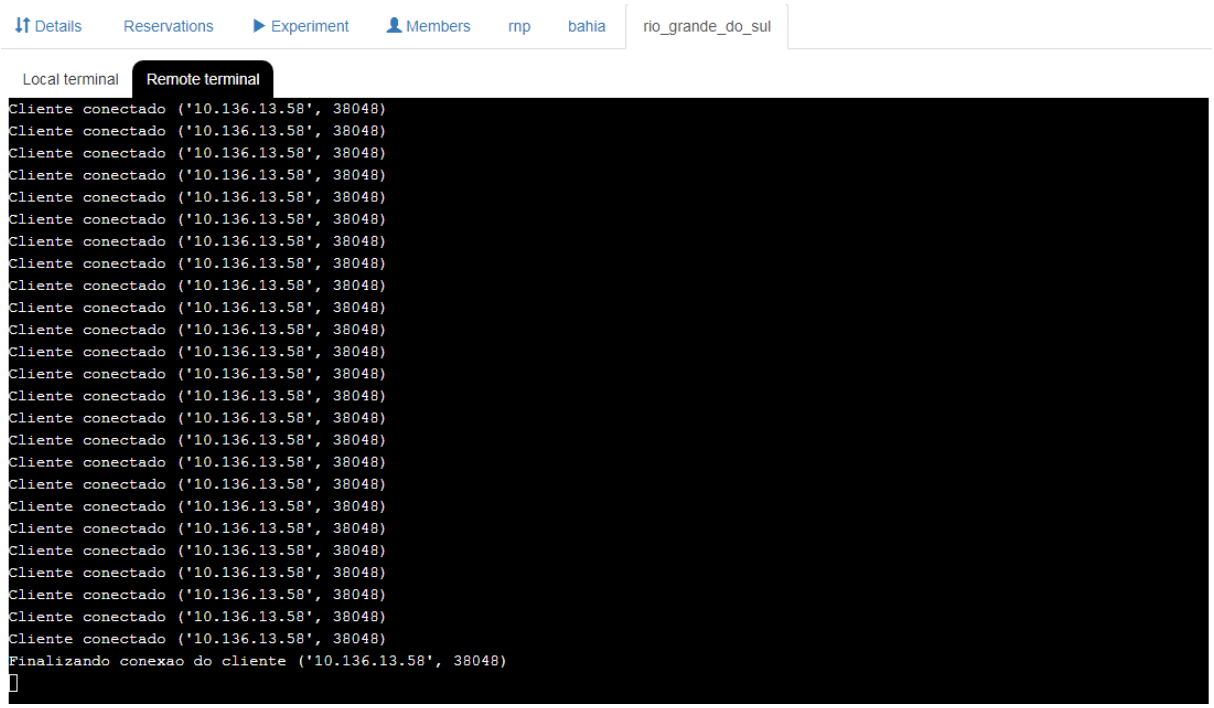
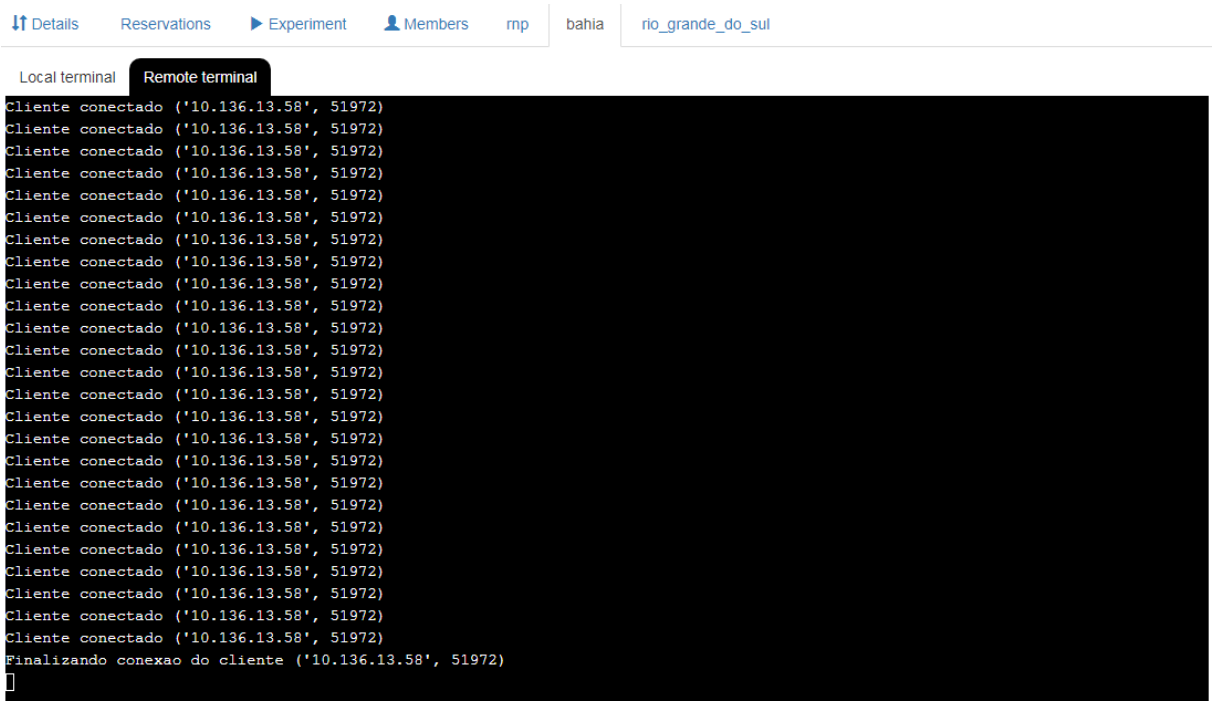


Figura 47 – (Sistema de Monitoramento) Resposta do servidor bahia ao encerrar o cliente



```
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Cliente conectado ('10.136.13.58', 51972)
Finalizando conexao do cliente ('10.136.13.58', 51972)
```

Para finalizar a execução dos servidores basta teclar **Ctrl+C** e os servidores serão finalizados. Da mesma forma como feito na seção **Comunicação Básica**, aqui deixamos um desafio para o leitor: encontre na internet algum programa capaz de estressar um computador propositalmente, tanto em processamento quanto em memória. Coloque esse programa de estresse para ser executado em segundo plano num dos servidores desse experimento e após repita o processo desse experimento, para observar o aumento dos valore recebidos para a máquina que está parrando pela sobrecarga dos recursos.

5 CONCLUSÃO

As competências desenvolvidas nesse tutorial, permitiram que o leitor experimentador adquirisse conhecimento teórico e prático sobre a camada de transporte com foco nos protocolos da internet. Então após a leitura e do seguimento dos passos propostos por esse material, o leitor terá entendido a reconhecer a diferença entre o modo como se dá a comunicação via **TCP** e via **UDP**, além de formar o senso sobre a melhor opção de utilização de acordo com a aplicação que se deseja implementar.

Neste trabalho, também é notório a importância que deve-se dar às estruturas como o **testbed FIBRE** que permitem experimentar uma série de ambientes disponibilizando recursos computacionais conectados à Internet. Isso traz o benefício de aproximar a experimentação do cenário real de uma aplicação de rede, contribuindo para o bom desenvolvimento de projetos. Além disso, esse tutorial tem como finalidade fazer com que o leitor esteja mais familiarizado com o ambiente do **testbed FIBRE** para utilizá-lo sempre que necessário, uma vez que a maior utilização da plataforma por usuários que estejam dispostos a contribuírem com *feedbacks* implicará na melhoria da prestação desse serviço para a comunidade acadêmica.

6 AGRADECIMENTOS

Primeiramente, agradeço à Rede Nacional de Pesquisa que viabilizou a execução deste tutorial investindo para a boa sua elaboração através da disponibilidade de recursos financeiros e materiais pela "Chamada Interna - Tutorial de utilização do testbed FIBRE". Posteriormente, agradeço à instituição Universidade Federal de Minas Gerais, que através do Wireless Network Laboratory, possibilitou que este trabalho chegasse até mim. Por último, gostaria de agradecer ao Centro Federal de Educação Técnológica de Minas Gerais, o qual fomentou em mim o desejo de aprendizagem em Redes de Computadores no ensino técnico, fato que até hoje resulta em frutos na minha carreira acadêmica.

REFERÊNCIAS

KUROSE, J. F.; ROSS, K. W. **Computer Networking: A Top-Down Approach (6th Edition)**. 6th. ed. [S.l.]: Pearson, 2012. 166 p. ISBN 0132856204, 9780132856201.

KUROSE, J. F.; ROSS, K. W. **Computer Networking: A Top-Down Approach (6th Edition)**. 6th. ed. [S.l.]: Pearson, 2012. ISBN 0132856204, 9780132856201.

O que é FPS? Confira significado e games de sucesso no competitivo. **E-SPORTV**, Rio de Janeiro, 16 jun. 2019. Disponível em: <https://sportv.globo.com/site/e-sportv/noticia/o-que-e-fps-confira-significado-e-games-de-sucesso-no-competitivo.ghtml>. Acesso em: 5 nov. 2019.

TANENBAUM, A. **Computer Networks**. 4th. ed. [S.l.]: Prentice Hall Professional Technical Reference, 2002. ISBN 0130661023.

ANEXO A – COMUNICAÇÃO BÁSICA - UDP CLIENT

```
1 from socket import *
2 import time
3
4 SERVER_IP = '127.0.0.1'
5 SERVER_PORT = 12000
6 SERVER = (SERVER_IP, SERVER_PORT)
7 udp_client_socked = socket(AF_INET, SOCK_DGRAM)
8
9 print('Socket cliente em execucao\nPara sair use CTRL+X')
10 print('=== Digite caracteres minusculos: ')
11 send_msg = input()
12 received_msg = None
13 while send_msg != '\x18':
14     udp_client_socked.sendto(send_msg.encode(), SERVER)
15     time.sleep(0.5)
16     received_msg, _ = udp_client_socked.recvfrom(2048)
17     print('=== Resposta do servidor: \n', received_msg.decode('utf-8'), '\n
       ')
18     print('=== Digite caracteres minusculos: ')
19     send_msg = input()
20
21 udp_client_socked.close()
```

ANEXO B – COMUNICAÇÃO BÁSICA - UDP SERVER

```
1 from socket import *
2
3 SERVER_IP = ''
4 SERVER_PORT = 12000
5 ORIGIN = (SERVER_IP, SERVER_PORT)
6
7 serverSocket = socket(AF_INET, SOCK_DGRAM)
8 serverSocket.bind(ORIGIN)
9 print('Servidor em execucao')
10 while 1:
11   message, clientAddress = serverSocket.recvfrom(2048)
12   message = message.decode('utf-8')
13   print(clientAddress, message)
14   modifiedMessage = message.upper() + ' - OK'
15   serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

ANEXO C – COMUNICAÇÃO BÁSICA - TCP CLIENT

```
1  import socket
2  HOST = '127.0.0.1'      # Endereco IP do Servidor
3  PORT = 5000             # Porta que o Servidor esta
4  tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5  dest = (HOST, PORT)
6  tcp.connect(dest)
7  print ('Para sair use CTRL+X\n')
8  msg = input()
9  while msg != '\x18':
10     tcp.send (msg.encode())
11     msg = input()
12 tcp.close()
```

ANEXO D – COMUNICAÇÃO BÁSICA - TCP SERVER

```
1 import socket
2 HOST = ''                # Endereco IP do Servidor
3 PORT = 5000              # Porta que o Servidor esta
4 tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5 orig = (HOST, PORT)
6 tcp.bind(orig)
7 tcp.listen(2)
8 while True:
9     con, cliente = tcp.accept()
10    print ('Concetado por', cliente)
11    while True:
12        msg = con.recv(1024)
13        if not msg: break
14        print (cliente, msg.decode('utf-8'))
15    print ('Finalizando conexao do cliente', cliente)
16 con.close()
```

ANEXO E – MONITOR DE RECURSOS - CLIENT

```
1 import socket
2 import sys
3 HOST = sys.argv[1] #IP do servidor obtido por arugmento de execucao
4 PORT = 5000            # Porta que o Servidor esta
5 tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6 dest = (HOST, PORT)
7 tcp.connect(dest)
8 while True:
9     tcp.send (' '.encode())
10    modifiedSentence = tcp.recv(1024)
11    print(modifiedSentence.decode('utf-8'))
12 tcp.close()
```

ANEXO F – MONITOR DE RECURSOS - SERVER

```
1 import psutil
2 import socket
3 import time
4
5 HOST = ''                # Endereco IP do Servidor
6 PORT = 5000              # Porta que o Servidor esta
7 tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8 orig = (HOST, PORT)
9 tcp.bind(orig)
10 tcp.listen(1)
11 while True:
12     con, cliente = tcp.accept()
13     print ('Concetado por', cliente)
14     while True:
15         msg = con.recv(1024)
16         if not msg: break
17         print ("Cliente conectado",cliente)
18         time.sleep(3)
19         send_msg = 'cpu percent: '+str(psutil.cpu_percent())+'\
20             - memory usage percent: '+ str(psutil.virtual_memory().percent)
21         con.send(send_msg.encode())
22     print ('Finalizando conexao do cliente', cliente)
23     con.close()
```