INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA SUL-RIO-GRANDENSE - IFSUL, *CAMPUS* PASSO FUNDO.
CURSO DE TECNOLOGIA EM SISTEMAS PARA INTERNET

THIAGO DOS SANTOS MARINI

COMPARATIVO DA COMUNICAÇÃO DE DADOS EM DISPOSITIVOS MÓVEIS: *WEB SERVICES* E *SOCKETS.*

Prof. José Antônio Oliveirá de Figueiredo

PASSO FUNDO, 2012

THIAGO DOS SANTOS MARINI

COMPARATIVO DA COMUNICAÇÃO DE DADOS EM DISPOSITIVOS MÓVEIS: *WEB SERVICES* E *SOCKETS*.

Monografia apresentada ao Curso de Tecnologia em Sistemas para Internet do Instituto Federal Sul-Rio-Grandense, *Campus* Passo Fundo, como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet.

Orientador (a): Prof. José Antônio Oliveirá de Figueiredo

PASSO FUNDO, 2012

THIAGO DOS SANTOS MARINI

COMPARATIVO DA COMUNICAÇÃO DE DADOS EM DISPOSITIVOS MÓVEIS, *WEB SERVICES* E *SOCKETS*

Trabalho de Conclusão de Curso aprovado em 13/12/2012 como requisito parcial para a obtenção do título de Tecnólogo em Sistemas para Internet

Banca Examinadora:

_____

Prof. Esp. José Antônio de Oliveira de Figueiredo

_____

Prof. Me. Anubis Graciela de Moraes Rossetto

_____

Prof. Me. Lisandro Lemos de Machado

_____

Prof. Me. Evandro Miguel Kuszera
Coordenação do Curso

PASSO FUNDO, 2012

“Meus filhos terão computadores, sim, mas antes terão livros.
Sem livros, sem leitura, os nossos filhos serão incapazes de escrever,
inclusive a sua própria história.”
Bill Gates

Este trabalho que foi desenvolvido ao longo de um grande período e tendo uma importância fundamental para minha vida pessoal, então gostaria de dedicar ao meu pai meu maior apoiado, a minha mãe, meus avós, pessoas fundamentais para minha vida e de um modo geral a toda minha família e aos meus amigos que sempre estiveram presente na minha vida me dando forças.

RESUMO

O presente trabalho busca comparar os métodos de transmissão de dados *Web Service* e *Socket*, visando os aspectos de desempenho em tráfego de pacotes na rede e o tempo de resposta da comunicação proposta em dispositivos móveis. Para obter o resultado dos parâmetros foram desenvolvidas duas aplicações, um servidor e dois clientes móveis com a mesma funcionalidade, cada um utilizando uma das tecnologias de comunicação a serem comparadas. A plataforma móvel usada utilizada como cliente possuía o sistema operacional Android, entretanto para os servidores utilizaram-se as linguagens de programação Java e Java web. A funcionalidade da aplicação foi pensada para conseguir obter vários comportamentos das aplicações, na finalidade, aumentar os parâmetros de comparativos entre os dois métodos. A aplicação foi desenvolvida para que o cliente *Android* gere uma quantidade de números aleatórios, definidos pelo usuário, os quais são mandados para o servidor. O servidor após receber os valores realiza o cálculo de média, logo em seguida retorna esse resultado para o cliente. A coleta dos pacotes que foram transmitindo na rede foi realizada com o software *Wireshark*, obtendo o número de pacotes trocados somente entre o cliente e o servidor. Para a medição do tempo dos métodos foi utilizado um método que a linguagem Android suporta e calcula do início da execução até o final do método principal do cliente, obtendo assim o tempo de execução de toda aplicação. Tendo como base os resultados da quantidade de pacotes transferida e o tempo da execução realizou-se uma análise dos dados coletados, assim gerando tabelas, gráficos e porcentagem da diferença entre os métodos.

Palavras-chave: Dispositivos Móveis, Android, *Web Services*; *Socket*; redes.

ABSTRACT

This study aims to compare the methods of transmitting data and Web Socket Service, aiming performance aspects in packet traffic on the network and the response time of the proposed communication on mobile devices. For the outcome parameters were developed two applications, one server and two clients with the same mobile functionality, each using one of the communication technologies to be compared. The mobile platform used as a client had used the Android operating system for servers however were used programming languages Java and Java web. The functionality of the application is designed to achieve various behaviors of the applications in order to increase the parameters of comparing the two methods. The application was developed for the Android client manages a quantity of random numbers, user defined, which are sent to the server. The server after retrieving the values performs the calculation of average, then immediately returns this result to the client. The collection of packets that were transmitted in the network was performed with the Wireshark software, obtaining the number of packets exchanged only between the client and the server. For the time measurement method was used a method that language Android support and calculates the start of the execution until the end of the main method of the customer, thus obtaining the execution time of every application. Based on the results of the number of packets transferred and the time of the execution was carried out an analysis of the data collected thus generating charts, graphs and the percentage difference between the methods.

Key words: Mobile Devices, Android, *Web Service*; *Socket*; Network.

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A programação em rede de computadores está presente em diversos sistemas. Umas das diversas vantagens dessa tecnologia é a capacidade física e lógica, de distribuir operações computacionais passadas, com um grande volume de dados e informações entre máquinas diferentes, que possam estar separadas geograficamente e trabalhando em conjunto, havendo assim, a descentralização de tarefas de um mesmo sistema.

Umas das principais arquiteturas de redes, que começou a ganhar espaço nos anos 90, foi o modelo baseado em programação distribuída, que faz a separação de uma aplicação. Aplicações que fornecem serviços a serem consumidos e aplicações que consomem esses serviços.

Essa arquitetura, que distribui uma aplicação entre diversos dispositivos ou simplesmente fornece um serviço, traz muitos benefícios, pois aumenta a confiabilidade da aplicação e auxilia na redução dos custos. Assim não é mais preciso investir em um único hardware e as confiabilidades das aplicações aumentaram, mas pelo fato que mesmo havendo falha durante o processo de uma máquina o sistema não seria interrompido.

Cliente e servidor possuem aplicativos que quando executados em diferentes máquinas, podem trocar informações por meio da rede de computadores.

Aplicações clientes são capazes de consumir serviços quando conectados com o servidor, o cliente (consumidor), para que seja feita a conexão bem sucedida deve saber quem será o fornecedor dos serviços ou processo a ser consumido, sabendo o endereço. Um ponto importante que tanto o consumidor quanto quem vai disponibilizar um serviço, deve utilizar o mesmo protocolo de comunicação para que seja possível que a ligação entre dispositivos seja realizada.

Os objetos de estudo neste trabalho são *Web Services* e *Sockets.* Ambos compõem uma arquitetura de sistemas em rede, pois, irão conectar e disponibilizar conexões entre dispositivos que não estão conectados, viabilizando assim a comunicação (troca de informações) entre aplicações ou até mesmo uma parte da aplicação que pode ser chamada de processo.

Este estudo busca conhecer as diferenças entre os métodos de comunicação e documentar essas diferenças, para que possam existir parâmetros quanto o uso dessas

tecnologias nos dispositivos móveis, especificamente na plataforma Android. Destacando um método com melhor desempenho na rede, em transmissão de pacotes e tempo de resposta na comunicação em uma comunicação entre dispositivos móveis com máquinas Desktop. No intuito de coletar informação para que possam ser mais bem detalhado os métodos de comparação, foram pensados e elaboradas duas aplicações com as mesmas funcionalidades para os dois métodos de transmissão de dados via rede, assim conseguindo detalhar alguns tópicos que foram levados em consideração, como: O desempenho da rede em número de pacotes transmitidos, tempo de resposta entre o cliente e o servidor e o detalhamento da programação que foi desenvolvida em cada programa para cada método.

1.1 MOTIVAÇÃO

A motivação do trabalho consistiu em realizar um estudo onde os resultados obtidos possam contribuir para resolver algumas dúvidas que são sobre diferenças de desempenho entre os métodos de transmissão de dados *Socket* e *Web Services,* as quais foram levantadas em um projeto de pesquisa realizado antes do trabalho de conclusão. Esses questionamentos não estavam claros, muitas vezes por falta de documentação ou até mesmo por serem demonstra dos poucos resultados de testes nas literaturas.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Pesquisar e realizar o comparativo sobre as tecnologias (métodos) de comunicação de troca de dados *Web Services e Socket,* analisando desempenho na troca de pacote na rede.

1.2.2 Objetivos específicos

- Utilizando meios de pesquisa, elaborar uma revisão bibliográfica, buscando aperfeiçoar o conhecimento teórico sobre os métodos citados.
- Criar um aplicativo que será implementado nas tecnologias *Web Services e Socket,* visando o conhecimento de ambos os métodos na questão de comunicação entre dispositivos móveis com máquinas normais.

- Definir parâmetros de testes, realizar a captura dos pacotes da rede, usando um *software*, e medir o tempo de resposta do inicio da execução até o final.
- Executar os testes e fazer a análise dos testes.

2 WEB SERVICES

Web Services traz consigo uma solução para fazer a comunicação de dados na rede. A solução foi criada para proporcionar e facilitar a comunicação entre aplicações, assim sendo desatrelada de linguagens de programação e facilitando a criação de novos serviços na internet.

> Uma definição bastante simples para Web Services: São aplicações que aceitam solicitações de outros sistemas através da Internet. Web Services são interfaces acessíveis de rede, para as funcionalidades da aplicação, que utilizam em sua construção tecnologias padrões da Internet. Através dessas afirmações, observa-se que Web Services são serviços que visam facilitar o processamento distribuído em sistemas heterogêneos. Estes serviços são baseados em um conjunto de padrões da Internet definidos pelo W3C. Fonte: Menéndez (2002),

2.1 O QUE SÃO *WEB SERVICES*

Web Services nada mais é que uma funcionalidade que será acessada por um sistema e não diretamente por um usuário.

Podemos considerar como uma solução para integrações entre sistemas de aplicações, consumindo ou disponibilizando serviços, podendo interligar vários sistemas diferentes. Esta tecnologia possibilita que novos sistemas interajam com sistemas já existentes, permitindo que sistemas desenvolvidos em plataformas diferentes possam se comunicar de forma simples e rápida, não importando se estão separados geograficamente. Existem duas abordagem de *Web Services*: REST e SOAP.

Web Services é um padrão adotado pela W3C[1] e também pela indústria por ter passado por inúmeros níveis de maturidade que a W3C submete para ganhar sua certificação. Dentre eles a abordagem SOAP faz o uso do formato XML[2] para troca de dados entre aplicação,

---

[1]Consórcio internacional, e que visa desenvolver padrões para a criação e a interpretação de conteúdos para a Web.

[2]XML (*Extensible Markup Language*) é uma linguagem de marcação. Seu propósito principal é a facilidade de compartilhamento de informações através da internet.

possibilitando o envio e o recebimento de mensagens. Contudo segundo MORO, Tharcis D.; DORNELES, Carina F.; REBONATTO, Marcelo T.: (2011) na abordagem REST é possível representar os dados usando no formato JSON (*JavaScript Object Notation*) ao invés de somente XML.

Segundo COULOURIS (2007, p.670) “O crescimento da web nos últimos cinco anos, prova a eficácia do uso de protocolos simples para internet, com base em um grande número de serviços e aplicação remotos.”, outro fator importante para o avanço no número de serviços seria o crescimento e o melhoramento no desempenho de dispositivos móveis, que estão muito mais rápidos e mais acessíveis.

De um ponto de vista técnico, *Web Services* constituem-se em softwares de baixo acoplamento, reutilizáveis, com componentes feitos para serem facilmente acessados pela internet. (ABINADER; ABILIO, 2006, p. 11).

O termo serviços web e servidor web não devem ser confundidos: “servidor web é um computador responsável por aceitar pedidos HTTP[1] de clientes, geralmente os navegadores, e servi-los com respostas HTTP, incluindo opcionalmente dados, que geralmente são páginas web, tais como documentos HTML“ (http://pt.wikipedia.org/wiki/Servidor_web).

“*Web Services* é uma interface de Serviço Web que oferece um conjunto de operações que podem ser usadas por clientes na internet. As operações de um serviço web podem ser oferecidas por uma variedade de recursos diferentes, um exemplo, programas, objetos ou banco de dados” (COULOURIS, George; DOLLIMORE, Jean; KINDBERG, Tim. 2007 p.674). Outro exemplo de um serviço web poderia ser um sistema que está sendo construído e irá conceder vendas pela internet, esse sistema terá que calcular o frete para entregá-lo até o cliente, mas sabendo que os correios já possuem um sistema que faz o cálculo do frete e está sendo disponibilizado no formato de *Web Services*, então é possível fazer seu sistema consumir esse serviço, por exemplo, passando por parâmetro a UF do cliente e o CEP e outras informações do produto exemplo peso tipo de embalagem tamanho que o cálculo do valor, do tempo frete será feito pelo *Web Services* dos correios.

2.1.1 Vantagens

Segundo ENIO PERPÉTUO, JÚNIOR IVERSON LOURENÇO JAGIELLOA (2003), Web Services são soluções para aplicações distribuídas na internet, suas principais vantagens são:

- Pode ter integração entre aplicações construídas com tecnologias diferentes.
- São utilizados padrões de códigos legíveis para o ser humano, isso facilita que novas aplicações sejam criadas com maior facilidade.

2.1.2 Desvantagens

Segundo ENIO PERPÉTUO, JÚNIOR IVERSON LOURENÇO JAGIELLOA (2003), Web Services são soluções para aplicações distribuídas na internet, suas principais desvantagens são:

- A aplicação que consumir muitos *Web Services* tem perda desempenho, devido a demora para encontrar cada serviço.

- A construção e integração de *Web Services* podem ser de alto custo, pois para que um *Web Services* conter uma boa performance e flexibilidade é necessário uma programação mais avançada com pessoal com maior conhecimento.

2.2 PRINCIPAIS CARACTERÍSTICAS DOS SERVIÇOS WEB

Dentre as duas abordagens de *Web Services,* SOAP e REST, o presente trabalho buscou aprofundar-se e posteriormente implementar um dos clientes móveis da aplicação utilizando a abordagem SOAP, devido ser um método seguro, pois não permite que os dados

fiquem visíveis na hora da transmissão. Se comparando com o método REST, que manda todas suas informações pela url, assim deixando visíveis os dados transmitidos na rede.

2.3 SOAP

SOAP é um protocolo baseado em XML para troca de informação entre computadores. Este protocolo facilita a conexão de serviços e métodos remotos que são disponibilizados para uma aplicação-cliente. Assim, uma aplicação-cliente pode adicionar um Serviço Web disponibilizado por alguma empresa.

Mensagens SOAP são escritas em XML, por isso tornam-se independentes de linguagens e plataformas. Desta forma, um cliente PHP usando SOAP, rodando em Linux, ou cliente Java rodando em Windows[3], podem se conectar em um servidor Unix executando SOAP. Outro exemplo que seria fácil de explicar é o aplicativo *mobile* do Facebook, que uma vez instalado no aparelho celular, que tenha suporte a esse, então irár desfrutar de serviços prontos, e não precisará acessar a pagina do Facebook em um navegador, sim abrindo o aplicativo instalado. “Protocolo SOAP faz a utilização do XML como esquema de representação de dados de requisição e resposta. O protocolo SOAP utiliza HTTP para fazer suas trocas de mensagens, mas hoje em dia podem ser utilizados vários protocolos de comunicação incluindo SMTP, TCP ou UDP” (COULOURIS, GEORGE, DOLLIMORE JEAN, KINDBERG TIM, 2007, p.673).

A sua especificação declara que quem tem a responsabilidade de representar o conteúdo de mensagens individuais é o XML, segundo Coulouris “As regras sobre os destinatários das mensagens devem processar os elementos XML que elas contêm. Como HTTP e SMTP devem ser usados para comunicar mensagens SOAP. É esperado que as versões futuras da especificação definam como usar outros protocolos de transporte, por exemplo, TCP”.

“Os serviços web são projetados para suportar a computação distribuída na internet, na qual são usadas muitas linguagens de programação diferentes. Eles são independentes de

[3] Marca registrada Microsoft.

qualquer paradigma de programação em particular” (, GEORGE, DOLLIMORE JEAN, KINDBERG TIM, 2007, p 676).

O programador que desenvolve uma aplicação que irá consumir um serviço web não precisa se preocupar com as linguagens específicas. API's SOAP foram implementadas em muitas linguagens de programação, incluindo Java, Java Script, Perl, Pyton, .Net, C,C++,C# e Visual Basic. Um dos fatores que possibilitam essa facilidade é que o protocolo SOAP usa XML para representar mensagens e HTTP para fazer o transporte.

2.3.1 Representação de Mensagens SOAP

As representações de mensagens do protocolo SOAP são também o protocolo que é utilizado para o transporte das mensagens SOAP. São representados em formato textual auto descritivo, (XML). Essa representação ocupa mais espaço que se comparado com um formato binário, e por sua vez exige mais processamento, mas trás como vantagem um formato legível para os seres humanos, facilitando a compreensão das mensagens e a facilidade de depuração do arquivo.

Web Services geralmente fornece uma descrição, na qual estão incluídas informações de um serviço para a aplicação, como a URL do servidor. Isso ocorre devido ao entendimento baseado em um sistema cliente e servidor.

2.3.2 Mensagens SOAP

Para entender o conceito de mensagens SOAP precisamos pensar em um envelope, que no contexto do protocolo SOAP vai existir um cabeçalho que é opcional e um corpo para o envelope. O cabeçalho das mensagens contém um contexto necessário para um serviço ou também pode ser usado para manter *log* ou auditorias das operações. É possível segundo COULORIS, que um intermediário possa interpretar e atuar sobre as informações presentes nos cabeçalhos das mensagens, por exemplo, adicionando, alterando ou removendo informações.

Uma mensagem SOAP transporta um documento XML para um serviço web em particular. Uma mensagem SOAP foi desenvolvida para suportar comunicação Cliente-Servidor, assim também sendo usada para transmitir um documento.

O corpo do envelope SOAP é muito importante, pois é nele que um documento a ser comunicado está localizado, nele colocado as referências a um esquema XML, que é importante, por que trás consigo a descrição do serviço a qual tem a função de definir os nomes e tipos usados no documento.

Esse tipo de mensagem SOAP pode ser enviado de forma síncrona ou de forma assíncrona. Na figura 1 podemos ver uma ilustração como seria um Envelope SOAP.

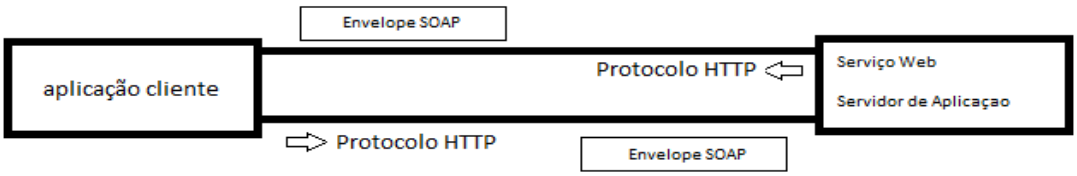
Figura 1: Envelope das mensagens SOAP.

Fonte: W3C.

2.3.3 Transporte SOAP

Em SOAP, o transporte é independente do envelope, pois o envelope não contém nenhuma referência ao endereço de destino. O transporte então fica a cargo de um protocolo de transporte desejado, podendo ser HTTP, SMTP, TCP ou UDP, que vão ter a responsabilidade de especificar o endereço de destino. A figura 2 está ilustrando a transmissão de um envelope SOAP.

Figura 2: Transporte das mensagens SOAP



Fonte: do autor

2.3.4 WSDL

O WSDL (*Web Service Description Language*) é uma linguagem de descrição de serviços de web tendo como objetivo dispor de um formato padronizado os serviços disponíveis para serem consumidos.

Conforme Weerawarana, et al. (2005), esta linguagem de descrição foi criada no ano 2000, originada da combinação de duas linguagens: NASSL (Network Application Service Specification) da IBM e SDL (Service Description Language) da Microsoft. No ano seguinte a versão 1.1 foi enviada como nota para W3C e em 2007 tornou-se recomendação em sua versão 2.0. A linguagem WSDL é considerada um vocabulário XML utilizado para descrever e localizar Web services. Permite que desenvolvedores de serviços disponibilizem informações importantes para a utilização dos mesmos. É altamente adaptável e extensível, o que permite a descrição de serviços que se comunicam por diferentes meios, tais como SOAP, RMI/IIOP.

2.4 COMUNICAÇÃO DE *WEB SERVICES*

Em geral, os serviços web adotaram como padrão o uso de requisição e respostas síncronas, mas também podem se comunicar por meio de mensagens assíncronas, que podem ser usadas quando a requisição exige resposta. Exemplo: O cliente envia uma requisição e posteriormente recebe a resposta de forma assíncrona.

Os serviços web utilizam uma URI[4], a qual é utilizada para as referência do serviço. Quem faz o serviço direto é a URL[5] que em seu padrão contém o nome do domínio e de uma máquina na rede que sempre será acessada pelo serviço solicitante. Essa máquina na rede pode propriamente executar o serviço web ou outro computador servidor pode executá-lo.

O responsável pelo envio de mensagens SOAP é o comando *post* do HTTP. O servidor contendo uma URL que especifica um endereço de destino tem a função de mapear a URL para a implementação do serviço e ativar a plataforma e o código onde o serviço irá executar. O nome do método chamado está localizado no corpo do cabeçalho.

O *Web Services* não possui um protocolo que envie mensagens confiáveis na presença de falhas. O protocolo de comunicação mais usado é o HTTP, executado sobre o TCP[6], mas não traz uma segurança muito grande, pois ele em caso de atraso na confirmação de entrega e tenta retransmitir uma quantia determinada, mas se ainda assim não conseguir receber confirmação, ele da à conexão como encerrada.

2.5 TRENSPARENCIA DA TECNOLOGIA

Uma das características mais importantes de um sistema distribuído é ser o mais transparente possível para os usuários da aplicação, que irão desfrutar de serviços que de alguma forma não estarão sendo executados em sua máquina ou dispositivo móvel. Um exemplo pode ser um aplicativo para dispositivos móveis que irão ser baixados em seus aparelhos que de uma forma irão buscar serviços via internet. Para o usuário isso fica transparente sem a alteração da funcionalidade da aplicação, esse exemplo mostrado é uma

---

[4]URI- uma cadeia de caracteres compacta usada para identificar ou denominar um recurso na Internet.

[5]URL- *Localizador-Padrão de Recursos* é o endereço de um recurso impressora etc.), disponível em uma rede; seja a Internet

[6]TCP (***Transmission Control Protocol***) Verifica se os dados são enviados de forma correta, na sequência apropriada e sem erros, pela rede.

forma mais genérica de transparência que ocorre para o usuário. Pode-se considerar também a facilidades para o programador que uma vez que está fazendo o uso de um *middleware*[7] fica mais protegido dos detalhes de representação e do empacotamento dos dados.

[7] Middleware: é um programa de computador que faz a médiação entre software e demais aplicações.

3 SOCKET

"*Socket* é um método de comunicação e troca de mensagens entre processos. Pode-se que o *Socket* é um método bem antigo para a comunicação de processos em rede, ele nasceu no UNIX BSD, que é um Sistema Operacional UNIX desenvolvido pela Universidade de Berkeley, na Califórnia, durante os anos 70 e 80" (http://pt.wikipedia.org/wiki/Berkeley_Software_Distribution), mas devido ao sucesso desta "tecnologia", diversos outros sistemas operacionais a "herdaram". Outras versões do Unix também, incluindo o Linux, Windows e no Macintosh OS[8] também fazem uso deste sistema.

Socket, em uma visão mais técnica, é um mecanismo que faz uma ligação direta entre dois *hosts*, criando uma conexão e que a partir daí poderá fazer transferência de dados tantos sobre os protocolos TCP ou UDP[9].

Socket é uma técnica de programação que está na camada de transporte, e trabalha diretamente com os protocolos TCP ou UDP.

3.1 CONCEITOS BÁSICOS SOBRE A PILHA TCP/IP

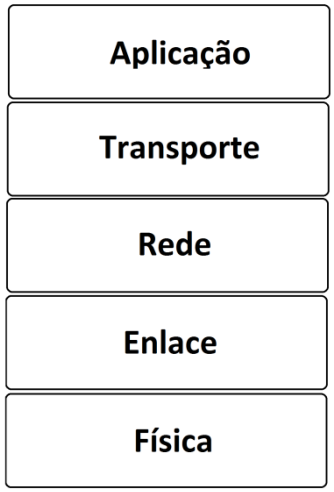
A pilha TCP/IP é como um modelo de camadas que carrega um conjunto de serviços. Nesse modelo de camadas quanto menor o nível de abstração, maior a complexidade, pois estará trabalhando mais perto do hardware de rede, por outro lado, quanto maior nível de abstração das camadas mais perto da API de usuário está, Exemplo, (Camada de aplicação). Existem cinco camadas na pilha TCP/IP, veja na figura 3.

---

[8] Marca registrada Apple.

[9]UDP (**User Datagram Protocol)** é um protocolo simples da camada de transporte, que não verifica se os dados são enviados ou chegam à sequência certa.

Figura 3: Cinco camadas da Pilha TCP/IP.



Fonte: do autor

3.2 CARACTERISTICAS DOS *SOCKET*S

Para conseguir fazer uma conexão cliente-servidor, o servidor terá que conter um *Socket* com uma porta dedicada para receber conexões, cada conexão tem um mecanismo que permite ao s receber varias conexões, criando vários canais ou outras conexões individuais para que cada cliente faça as suas trocas de dados.

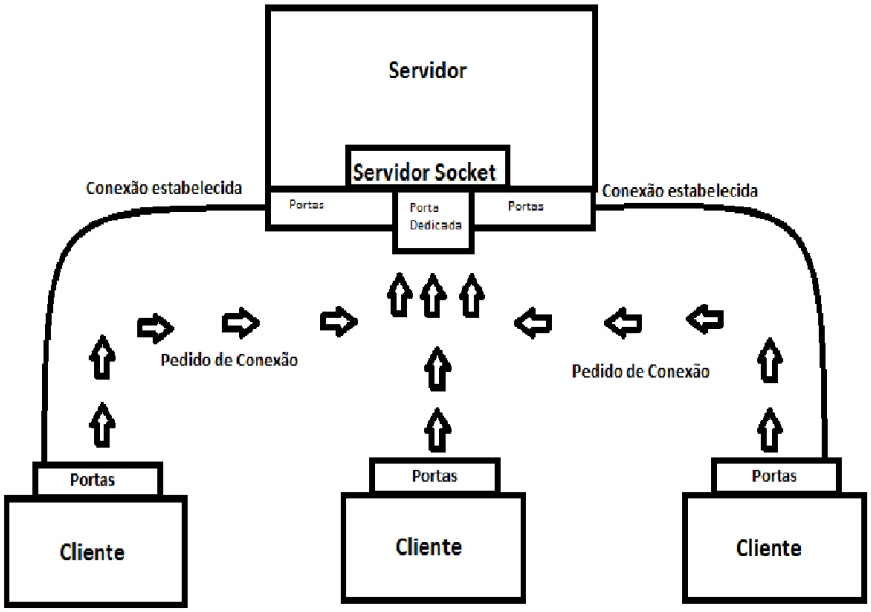
Por está situado na camada de transporte, ele serve de base para outros protocolos de comunicação de rede, que podem estar localizado em outras camadas, como por exemplo, o protocolo HTTP que está situado na camada de aplicação, que em algum momento quando executado irá executar um *Socket* para fazer a comunicação, mas isso fica transparente para o programador. Então de um modo geral qual quer aplicação que faz comunicação em rede irá em algum momento fazer o uso de s em seu contexto geral.

Na comunicação entre processo existe o *Server* que é um *Socket* diferenciado, é desenvolvido dentro da aplicação que recebe conexões dos clientes e cria *Socket* comum que irão fazer a comunicação de uma aplicação ou processo especifico. *Socket* comum é uma via de comunicação direta entre cliente e servidor na aplicação, que irá possibilitar a troca de dados.

Cada Socket é associado a um tipo de protocolo que pode ser TCP ou UDP. Então quando um *Socket* vai ser criado o protocolo de transporte já deve estar associado ao protocolo desejado.

Normalmente um servidor primeiro “escuta” e “aceita” um conexão e depois cria um novo processo para se comunicar com o cliente. Nesse meio tempo ele continuaria ”escutando” pedidos de conexão no processo original. (COULOURIS, GEORGE; DOLLIMORE JEAN; KINDBERG TIM. 2007 p. 159). A figura 4 representa como é feita à conexão entre cliente e servidor de uma forma bem simples.

Figura 4: Funcionamento geral das conexões entre *Socket*.



Fonte: do autor

Para fazer essa conexão o cliente executa alguns passos básicos:

1. Cria um *Socket*, usando a chamada de sistema *Socket*;
2. Conecta seu *Socket* ao endereço do servidor, usando a chamada de sistema *connect*;
3. Envia e recebem dados através do *Socket*, usando as chamadas de sistema *read* e *write*;

4. Encerra a comunicação, fechando o *Socket* através da chamada *close*;

E ainda nessa tecnologia cliente-servidor o Servidor também executa uma série de passos para oferecer seus serviços a seus clientes:

1. Cria um *Socket*, usando a chamada de sistema *Socket*;

2. Associa um endereço a seu *Socket*, usando a chamada de sistema *bind*;

3. Coloca o *Socket* em modo de escuta, através da chamada de sistema *listen*;

4. Aguarda um pedido de conexão, através da chamada *accept*(que gera um descritor específico para a conexão recebida);

5. Envia e recebem dados através do *Socket*, usando as chamadas de sistema *read* (ou *recv*) e *write* (ou *send*);

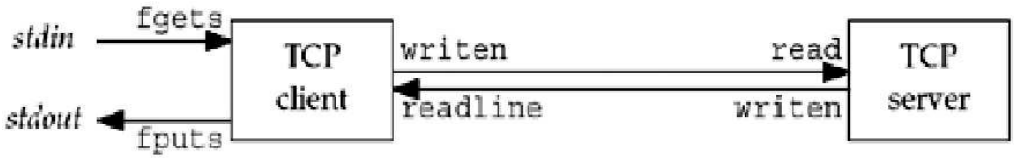
6. Encerra a comunicação com aquele cliente, fecha o descritor da conexão usando a chamada *close*;

7. Volta ao passo quatro, ou encerra suas atividades fechando seu *Socket* usando a chamada *close*. Isso de acordo com, (GEYER, Claudio; ROSINHA, Rômulo: 2006.)

Socket é um elo ou uma via bidirecional de comunicação entre dois programas ou dois processos, como propósito de fazer a ligação de redes de computadores juntamente com aplicativos. *Socket* também é uma abstração computacional que de uma forma direta e capaz de mapear uma porta de transporte (TCP ou UDP) juntamente com o endereço de rede.

Uns dos objetivos gerais do *Socket* é a comunicação entre processos na rede, que basicamente significa transmitir uma mensagem de um processo de uma determinada máquina para outro processo em outra máquina, que inclusive, poderá estar geograficamente separada. Para que essa via de conexão aconteça e as máquinas troquem mensagens, é preciso que o *Socket* contenha uma porta e um endereço IP, da máquina em que está sendo executado.

Figura 5: Comunicação entre *Socket*, sobre o protocolo TCP.



Fonte: Stevens, Richard W. ; Fenner, Bill; Rudoff, M. Andrew: Unix Network Programming.

A figura 5 está mostrando as chamadas de *Socket writen e read,* mas na programação pode-se usar outras chamadas *Socket* como a chamada *send e recv,* que fazem o mesmo papel, mas com um melhor controle sobre transmissão de dados.

O *Socket* de um cliente envia uma mensagem para um IP e uma ”porta” que será do Servidor, e só poderá fazer a conexão entre processos e receber as mensagens, se o *Socket* estiver associado a esse endereço IP e a essa porta.

Pode ser usado um *Socket* para enviar e receber mensagens, pois quando é estabelecida a conexão entre dois processos e criado uma via bidirecional, que como foi falado que pode receber e enviar usando essa via.

4 DESENVOLVIMENTO DAS APLICAÇÕES

Este capitulo apresenta os detalhes do desenvolvimento da duas aplicações com a mesma funcionalidade, mas implementadas com diferentes métodos de transmissão de dados, *Socket* e *Web Service*, tendo como base a arquitetura cliente e servidor. O servidor *Socket* foi desenvolvido em Java, já o servidor *Web Service* foi desenvolvido em Java Web. Os clientes foram desenvolvidos na plataforma Android, tendo o mesmo funcionamento.

Para realizar os testes propostos, uma simples aplicação cliente-servidor, foi desenvolvida, onde o cliente está rodando em um dispositivo móvel (Android) e gera uma quantidade de números aleatórios entre 0 e 999 (a quantidade de números gerados é informada pelo usuário). Estes números são enviados então para o servidor que recebe os valores e faz um simples cálculo de média aritmética entre os números gerados, devolvendo o resultado ao cliente que solicitou a operação.

Os testes de desempenho foram planejados de forma a obter as várias amostras do comportamento da rede, buscando “visualizar” o comportamento padrão de cada método. Para tentar obter este comportamento, foram coletadas 7 amostras para cada quantidade de números aleatórios gerada. Destas 7 amostras, foram descartadas 2 , a de maior valor e a de menor valor, este descarte foi feito para que se adquirisse um grau de confiança maior, já que observou-se que o aparelho móvel poderia estar executando alguma função em segundo plano interferindo assim no desempenho do resultado. Observou-se também que o primeiro teste de cada método era o que mais levava tempo para concluir, assim elevando significativamente a média geral dos demais testes. Os testes foram realizados com envio de 1.000, 10.000, 100.000, 1.000,000 números aleatórios. Verificou-se que no caso do *Web Services*, o maior número aleatório suportado por esse método foi 200.000, pois com quantidades maiores ocorre um erro de memória do aparelho móvel, causando o travamento da aplicação.

A programação dos dois métodos (*Web Services* e *Socket*) tem diferenças que influenciam no desempenho da aplicação e também no tempo de programação.

Socket por ser de uma camada de baixo nível, uma programação que trabalha em nível da camada de rede e transporte, necessita que seja trabalhado diretamente com IP e PORTA. Um ponto que deve ser ressaltado que em nível de programação o *Socket* transmite variáveis

simples, como, INT, STRING, DOUBLE etc; Nativamente, o *Socket*, não tem meios de enviar um *Array* ou um objeto pela rede.

Web Services de uma forma geral traz consigo inúmeras implementações que o permite trabalhar com *Array* e objetos, então o *Web Services* no momento que precisou mandar os números aleatórios do cliente para o servidor ele gera um envelope SOAP com todos os números gerados e manda esse envelope pela rede, o envelope não é enviado inteiro pela rede, ele que dividido para ser mandado pela rede para obter melhor desempenho, mas essa divisão é transparente para o usuário.

Para explicar as aplicações que foram usadas para a realização dos testes e a obtenção dos resultados irá ser mostrados trechos de código, para que possa ficar claro os testes realizados.

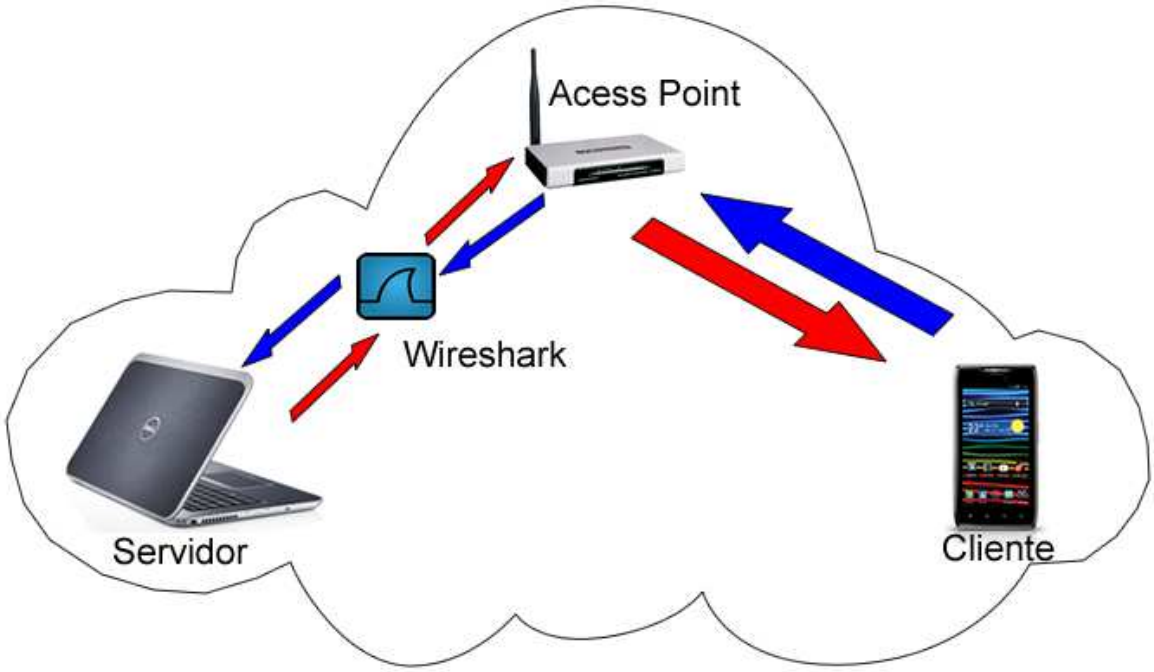
4.1 AMBIENTE DE TESTES

O cenário para execução dos testes foi o mesmo para cada aplicação. Foi usando como cliente, um dispositivo móvel, Motorola, RAZR MAXX, com plataforma Android, na versão 2.3.5, com 1GB de memória RAM, e um processador Dual Core de 1.2 GHz.

O servidor das aplicações foi rodado em um Ultrabook, DELL, Inspiron 14z, com sistema operacional Windows 7, home Premium 64 bits, 4GB de memória e um processador Intel I5 1.7GHz.

A rede utilizada foi uma comunicação Wireless 802.11g, 2.4 Ghz, em um Access Point exclusivo para realização dos testes, ou seja, não havendo interferência de outros dispositivos na rede de testes.

Figura 6: Ilustração do Cenário criado para os Testes.



Fonte: do autor

4.1.1 Aplicação *Socket*

A aplicação desenvolvida em *Socket* foi estruturada da seguinte forma; tando o cliente quanto o servidor Foram desenvolvidos na IDE Eclipse, mas na linguagem de programação para dispositivo móvel, Android.

O código completo da aplicação socket cliente e servidor estão nos anexos A e B respectivamente.

4.1.1.1 Servidor Socket

O Servidor *Socket* desenvolvido precisa ter uma porta padrão que ele sempre irá aguardar conexões nessa porta, no caso da aplicação desenvolvida a porta escolhida é 6500. A definição da porta na aplicação está listada no trecho de código abaixo na linha 34.

Figura 7: Configurando a porta na programação *Socket.*

```
33
34  ServerSocket serverSocket2 = serverSocket = new ServerSocket(6500);
35
```

Fonte: do Autor.

Uma vez que o servidor for executado, ele sempre estará aguardando solicitações de clientes, assim quando um cliente solicita conexão com o servidor, eles estabelecem um túnel, uma via bidirecional que possibilita pela porta que o *Socket* tem setada, a transmissão de pacotes ou dados na rede. Na linha 51 do código abaixo está mostrada a criação de um objeto *Socket* que criará uma conexão virtual, entre o Server *Socket* e o *Socket* cliente. Esse trecho de código fica encarregado de ordenar os bytes enviados ao servidor, que devem chegar na ordem certa.

Figura 8: Criando Objeto *Socket,* que realizara conexão com o cliente.

```
50
51  socket = serverSocket.accept();
52
```

Fonte: do Autor.

A aplicação foi desenvolvida no método que é baseado em conexão, um método seguro que usa o protocolo TCP. Segundo Hopson “A operação baseada em conexões emprega o TCP (*Transport Control Protocol*). Um *Socket* nesse modo de operação precisa se conectar ao destino antes de transmitir os dados. Uma vez conectados, os *Sockets* são acessados pelo uso de uma interface de fluxos: abertura-leitura-escrita-fechamento. Tudo que é enviado por um *Socket* é recebido pela outra extremidade da conexão, exatamente na mesma ordem em que foi transmitido. A operação baseada em conexões é menos eficiente do que a operação sem conexão, mas é garantida”, Como o *Socket* trabalha com a abertura-leitura-escrita-fechamento, sendo que é estabelecida, aberta a conexão com o cliente e depois é feita uma troca de leituras e escritas, o servidor “escreve uma variável na rede” e o cliente que está no outro lado tem que fazer a “leitura dessa variável” e vice versa, e assim quando não há mais informações para serem trocadas o *Socket* encerra sua conexão com o cliente que havia

solicitado conexão, mas o servidor logo após o fechamento da conexão, já está pronta para outra conexão.

A linha 59 da figura 9, do código do servidor, mostra o servidor recebendo, fazendo a “leitura” da variável que o cliente está mandando e guardando em uma variável local.

Figura 9: Variável recebendo do cliente a quantidade de números gerados.

```
58
59  int q = dataInputStream.readInt();
60
```

Fonte: do Autor.

O cliente manda a quantidade de números gerados, para que o servidor possa fazer a média, que nada mais é que a soma de todos os números que vem do cliente e dividir pela quantidade de números gerados, que o cliente está informando.

No servidor foi criado um laço de repetição, (FOR), para ler na mesma sequência todos os números que vem do cliente e poder processá-los. A figura 10 mostra o laço criado para ler os valores que o cliente está mandando para o servidor. No momento que o servidor lê um valor que vem do cliente ele adiciona esse valor a um *Array*, que irá conter todos os números que virão do cliente. No trecho de código abaixo, a linha 51 mostra o laço de repetição que tem como parâmetro a variável “q”, que nada mais é do que a leitura que o servidor fez da quantia de números gerados do cliente. A linha 52 mostra o servidor lendo os números aleatórios que estão sendo enviado pelo cliente e adicionando em um *Array*, a linha 53 mostra na console do servidor os números enviados pelo cliente.

Figura 10: Laço de repetição que recebe o que o cliente.

```
51          for (int i = 0; i < q; i++) {
52          nu.add(dataInputStream.readInt());// lendo os números aletórios que vem do cliente.
53          System.out.println("vet->"+ nu.get(i));// mostra no console os numeros criados pelo cliente
54          }
```

Fonte: do Autor.

Para realizar a média usou-se a lógica de somar todos os valores que foram adicionados no *Array* no servidor e logo após dividir pela quantia de números gerados, que o cliente manda para o servidor. Para a realização da soma foi utilizado Interator, que é segundo

o site javafree.uol.com.br “Interface que define as operações básicas para o percorrimento dos elementos da coleção”, que facilitou para realizar a soma de todos os números enviados pelo cliente. A figura 11 mostra o trecho de código da soma dos valores.

Figura 11: Realiza a soma e logo em seguida a média.

```
56          Iterator<Integer> it = nu.iterator();
57          double soma = 0;
58          while (it.hasNext()) {
59              soma += (Integer) it.next();  //realiza a soma dos números que vieram do cliente
60          }
61          double media = (soma/nu.size()); //realiza a média sobre os números aleatório
62
63          System.out.println("Média = " + media);// mostra no console a média
64
65          dataOutputStream.writeDouble(media);//mandando para o cliente o resultado da média aplicada
```

Fonte: do Autor.

Para a realização da divisão de todos os valores, pegou-se o resultado da soma e dividiu-se pelo tamanho do *Array*, (linha 61), pois ele tem o mesmo número de elementos que o cliente enviou.

4.1.1.2 Cliente Socket

O código da figura 12 mostra o cliente *Socket* mandando ou “escrevendo” uma variável que recebeu do usuário para o servidor.

Figura 12: Cliente *Socket* mandando para o servidor a quantia de números gerados.

```
69
70  dataOutputStream.writeInt(quanti);
71
```

Fonte: do Autor.

O cliente *Socket* tem a função de gerar a quantia desejada de números aleatórios e estabelecer conexão com o servidor, pois o servidor deve ser referenciado pelo seu IP e porta. A porta deve ser a padrão que foi implementada no servidor, como foi mostrada acima, a função que o cliente executa para gerar os números aleatórios, está explicada no trecho de código da figura 13.

Figura 13: Criando *Socket* e referenciando o servidor

```
58
59  socket = new Socket("192.168.0.5", 6500);//Criando socket para referenciar o servidor IP/PORTA
60
```

Fonte: do Autor.

O cliente *Socket*, como já mencionado na seção 4.2 o não transmite *Array* ou objeto pela rede, a solução para transmitir os números gerados foi a de mandar criar um número e transmitir o mesmo fazendo o servidor ler os números na mesma sequência que o cliente está transmitindo. Na linha 71 da figura 14 está o objeto Java responsável por criar números aleatórios. Já na linha 72 da figura 14 os números estão sendo criados no intervalo de 0 a 999 e adicionados em um *Array*. Para gerar o número que foi digitado pelo cliente, foi um laço de repetição, (FOR), que irá executar até a quantia de números que o cliente solicitou, segue o trecho de código implementado na aplicação *Socket*.

Figura 14: Laço de repetição que gera números randômicos.

```
70  for (int i = 0; i < quanti; i++) {
71      Random r = new Random();// função gerar números aleatórios
72      num.add(r.nextInt(1000));// seta o numero do intervalo que será gerado  e o intervalo vai de 0 a 999
73      System.out.println("->"+ num.get(i) );//mosta na console os numeros gerados
74      dataOutputStream.writeInt(num.get(i));// está mandando para o servidor cada numero aleatório gerado, sendo enviado 1 de cada vez.
75      }
```

Fonte: do Autor.

4.1.2 Programação *Web Services*

A programação da aplicação *Web Services* utilizada para a realização dos testes de performance de tráfego de dados e tempo de resposta na rede teve suas peculiaridades, como já mencionado na seção 4.2. Também, ele trabalha na camada de aplicação nas camadas TCP/IP, estando posicionado em uma camada que trás várias implementações, que ajudam na programação e já são nativas. *Desta forma*, é uma solução bastante usada para integrações de

sistemas distintos, transmitindo dados por meio de envelopes SOAP, que por sua vez trabalha com XML que oferece diversos benefícios para essa abordagem.

A Escolha do protocolo que seria utilizado para Web Services teve como fator de ser uns do mais utilizados, sendo escolhida para o desenvolvimento da aplicação o protocolo SOAP.

Os testes primeiramente foram pensados em serem realizados com as quantias de 1.000, 10.000, 100.000, 1.000.000 de números aleatórios, mas no decorrer dos testes foi identificado uma limitação por parte da tecnologia do *Web Services*, *o qual* cria um envelope SOAP que contém o XML com as variáveis que serão transmitida, então conforme o cliente irá gerando números aleatórios são adicionado nesse envelope que é criado temporariamente na memória do dispositivo móvel, que por sua vez tem um tamanho limite e quando os testes tentaram gerar 1 milhão de números aleatórios, o envelope cresceu tanto que estourou a memória RAM do dispositivo que tem 1GB de memória RAM. Assim sendo, obrigou a aplicação a encerrar o aplicativo sem conseguir mandar os números gerados para o servidor. O código completo da aplicação *Web Service* cliente e servidor estão nos anexos C e D respectivamente.

4.1.2.1 Cliente Web Service

O cliente foi desenvolvido no Android, na versão 2.3.3 e foi utilizada a biblioteca KSOAP2 para a criação do *Web Services*. Pode-se observar que o cliente *Web Services* tem algumas semelhanças com o cliente *Socket*, pois o método de gerar os números aleatórios são os mesmos. O cliente para se comunicar com o servidor precisa de algumas informações para permitir que a conexão seja realizada com sucesso. A figura 15 abaixo mostra os parâmetros que identificam o servidor e o serviço que será utilizado no servidor. A linha 33 está mostrando o NAMESPACE, que é o caminho para chegar até o arquivo que contém o método que será usado, já a linha 34 identifica o nome do método a ser usado. A linha 35 mostra o *SOAP ACTION* do *Web Services*, A linha 36 está mostrando a identificação do servidor *Web Services* na rede, para realizar a conexão com sucesso esse parâmetro deve estar correto.

Figura 15: Parâmetros para conexão com o serviço e Servidor *Web Services*

```
32
33      private String NAMESPACE = "http://tcc.ifsul.com/";
34      private String METHOD_NAME = "media";
35      private String SOAP_ACTION = "http://tcc.ifsul.com/WS/mediaReques";
36      private String URL = "http://192.168.0.5:8081/ServerWS/WS?WSDL";
37
```

Fonte: do Autor.

O cliente manda para o servidor a quantidade de números aleatórios que irá ser gerada. Na linha 73 da figura 16, mostra o cliente enviando o valor que o usuário digitou que é a quantidade de números aleatórios desejada.

Figura 16: Cliente enviando para o servidor a quantia de números Gerados.

```
72
73  request.addProperty("v1",quanti);
74
```

Fonte: do Autor.

A função que é responsável por gerar os números aleatórios foi desenvolvida de forma idêntica ao método baseado em *Socket*, assim gerando a quantidade de números aleatórios que o cliente deseja, no intervalo de 0 a 999, assim representado na figura 17.

Figura 17: Trecho do código que gera os números aleatórios

```
76
77  for (int i = 0; i < quanti; i++) {
78  Random r = new Random();
79  num.add(r.nextInt(1000));
80  Log.d("marcador", num.get(i).toString() );
81  }
82
```

Fonte: do Autor.

O cliente *Web Services* consegue trabalhar na transmissão de dados na rede, mandando um Array ou Objeto. Por uma limitação da biblioteca KSOAP2 que não consegue transmitir ou serializar um *Array* de *Integer*, então a solução que foi usada teve que transformar o Array de Integer e String, que está sendo mostrada na figura 18 do na linha 85.

Figura 18: Cliente adicionando ao envelope SOAP a lista inteira de valores gerados pelo cliente.

```
84
85  request.addProperty("v2", num.toString());
86
```

Fonte: do Autor.

4.1.2.2 Servidor Web Service

O servidor foi desenvolvido em Java Web pela IDE NetBeans 7.2. O Server *Web Services* a estrutura padrão foi criada com o modo gráfico que o NetBeans proporciona para a criação de *Web Services*, então baseado na proposta da aplicação foi desenvolvido o método que recebe os dados gerados pelo cliente. Como o cliente *Web Services* transforma a lista em uma *String*, então para o servidor trabalhar com isso ele primeiramente não consegue identificar os valores separadamente, desta forma a solução utilizada foi quebrar a string em partes e separar os valores, posteriormente adicionando-os em um Array local no servidor, essa separação pode ser vista na figura 19.

Figura 19: Recebendo dados do cliente e separando os valores que nela estão.

```
29
30  String aux = v2.get(0).toString();
31  aux = aux.replace("[", "");
32  aux = aux.replace("]", "");
33  String[] a = aux.split(",");
34
```

Fonte: do Autor.

A linha 30, figura 19, está mostrando “v2” que é a variável do servidor que recebe o Array que vem do cliente, mas como foi explicado antes, no cliente é mandado um Array, mas como a biblioteca do KSOAP2 tem a limitação que não consegue nativamente mandar variáveis complexas pela rede. Então a solução encontrada era transformar o Array em String, que contém todos os valores que o Array, dividindo o fato então, o “v2” e atribuído para uma variável do tipo String local. Um exemplo do formato de String que vem do cliente seria assim: [1,2,3,4,5,6], então para realizar a média dos valores, foi preciso desmembrar a String,

na linha 31 e 32 está substituindo os colchetes por “nada” então assim deixando a String neste formato: 1,2,3,4,5,6, já na linha 33 estão adicionando todos os valores a um Array de String, assim pegando valor que está depois da “,” então voltando ao formato que no cliente inicialmente tinha sido adicionado, já tendo os valores gerados pelo cliente e assim podendo realizar a média dos mesmos.

A figura 20 mostra um laço de repetição que tem a função de exibir o conteúdo do Array de String que contém os valores enviados do cliente, que estão sendo mostrada na linha 37, já na linha 39 mostra o Array “nu”, que é um Array de *INTEGER*, recebendo os valores que estão contidos na STRING, pois essa troca de “tipo” de Array de uma string para um integer é necessária para a realização da média.

Figura 20: Laço de repetição em um Array de INT.

```
36
37  for (String exibe : a) {
38  System.out.println("Vet->" + exibe.trim() + "\n");
39  nu.add(Integer.parseInt(exibe.trim()));
40  }
41
```

Fonte: do Autor.

O cálculo da média foi realizado da mesma forma que no servidor *Socket*. Segue, na figura 21, o trecho de código para este cálculo.

Figura 21: Realização da soma de todos os valores.

```
42
43  Iterator<Integer> it = nu.iterator();
44  Double soma = 0.0;
45  while (it.hasNext()) {
46  soma += it.next();
47  }
48
```

Fonte: do Autor.

Aplicação do servidor *Socket* foi utilizado Interator, que ajuda no percorrimento dos valores que estão no Array, (linha 43). A soma está sendo feita em uma laço de repetição

“while”, que está representado na linha 45, e na linha 46 a cada ciclo do laço de repetição é feita a soma de número por número.

A média é realizada pegando a soma que está representada na figura 21, e dividindo pela quantidade de números que o usuário digitou na aplicação cliente. Pode-se observar essa operação na figura 22 que segue abaixo.

Figura 22: Realizando a média.

```
51
52  return Double.parseDouble(decimal.format(soma / nu.size()).replace(",", "."));
53
```

Fonte: do Autor.

Na figura 22 a média está sendo realizada diretamente no retorno que irá para o cliente, sendo que para fins de formatação do resultado que será enviado para o cliente é substituído ”,” por “.”, pois também na mesma linha está sendo formatado o resultado da média com dois dígitos depois do “.”. O responsável por essa formatação é o decimal.format, que está presente na linha 52 da figura 22.

5 COLETA DE DADOS

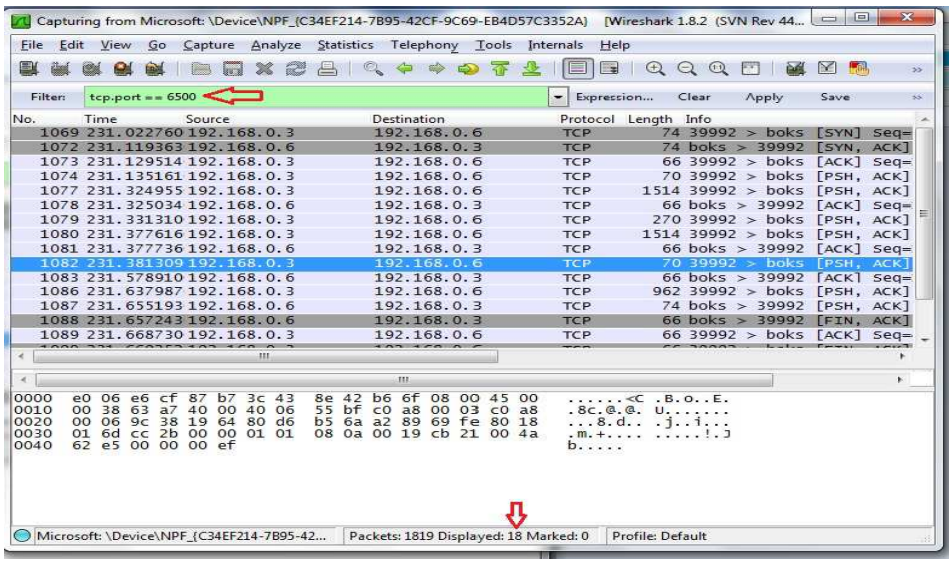
A coleta de dados tem a finalidade de colher variáveis dos dois métodos para que possam ter parâmetros para analisar de uma melhor forma os dois métodos.

5.1 IDENTIFICAÇÃO DOS PACOTES SOCKETS

As identificações dos pacotes foram feitas por um *Sniffer* de rede, que segundo Bradley Mitchell Os *sniffers* de rede monitoram o fluxo de dados na rede de computadores. Pode ser um programa ou um dispositivo de *hardware* com um software embarcado. As vezes chamados de sondas da rede ou bisbilhoteiros, assim fazendo uma copia dos dados trafegados na rede, mas sem redirecionar ou altera-los.

Para a análise do tráfego na rede foi utilizado o software *Wireshark.* Esse programa é capaz de identificar todos os pacotes que trafegam em uma máquina que esteja instalada. Para a utilização desse software precisa saber precisamente qual a “porta” e o protocolo de transmissão de dados como TCP ou UTP, que será utilizada para a comunicação entre cliente e servidor. No caso da programação *Socket*, a porta é definida no servidor e no software desenvolvido a 6500, pois nessa porta que serão executadas os serviços propostos, e no casso da aplicação também a realização da média da quantidade de números aleatórios que o cliente mandar para o servidor. Então sabendo a porta que a aplicação usa, é filtrada a porta e identificados somente os pacotes que estão sendo trocados entre cliente e servidor. Podemos observar na figura 23 que a primeira flecha vermelha que está localizada no canto superior da imagem, indica a porta e o protocolo que será usado pelo cliente e também pelo servidor, já no canto inferior onde a flecha vermelha está apontando para baixo e identificado o número de pacotes que foi capturado pelo filtro realizado, que é somente do cliente e do servidor.

Figura 23: A figura está mostrando coleta de pacotes Wireshark.



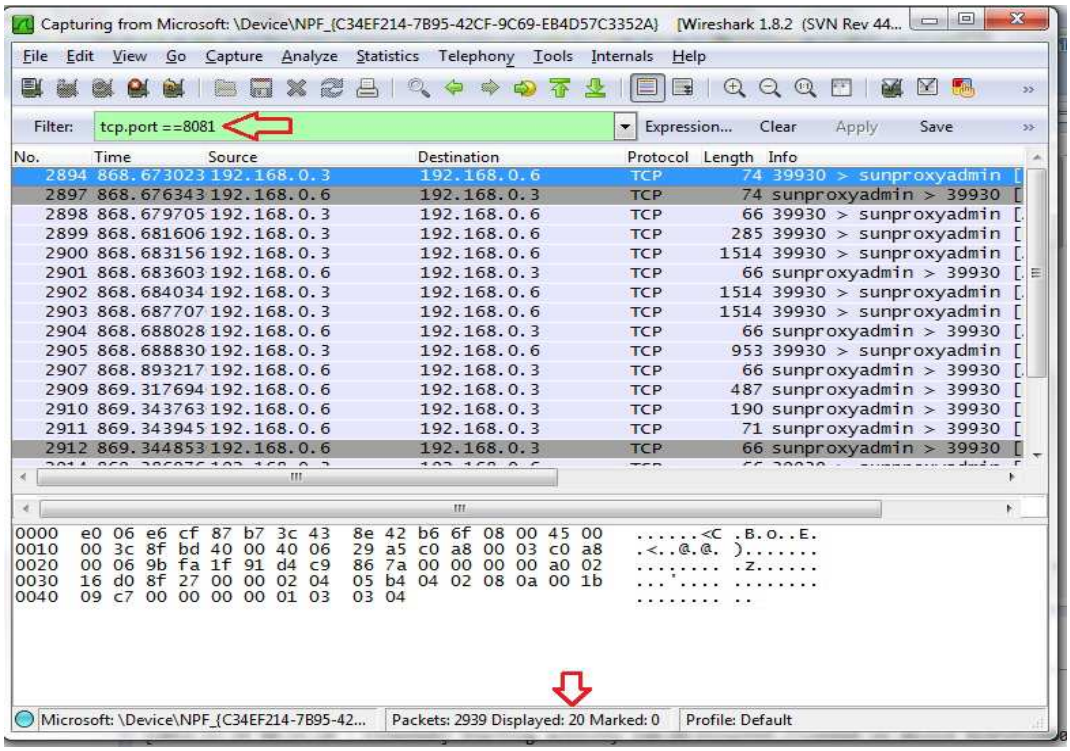
Fonte: do Autor.

5.2 IDENTIFICAÇÃO DOS PACOTES WEB SERVICE

No *Web Service* o servidor foi desenvolvido em Java Web que precisa de um servidor de aplicação para rodar suas aplicações, O servidor de aplicação usado foi Glassfish[10]. O servidor de aplicação tem uma porta padrão que pode ser vista na sua própria documentação, a porta padrão é a 8080. Alguns testes foram realizados no IFSUL que usa um serviço de Proxy na porta 8080. O quando eram realizados os testes e filtrado pela porta 8080 o Wireshark não coletava somente os pacotes que vinham do cliente e do servidos, mas também do servidor Proxy da instituição, (na mesma porta), então passou-se a usar a porta 8081 no servidor que, não continha nem um serviço sendo executado nela. A porta do *Web Service* então não era configurada manualmente como no servidor, pois quem é responsável pela porta é o servidor de aplicação, mas tem que ser referenciada no cliente da aplicação.na figura 24 mostra o filtro no Wireshark para identifica os pacotes TCP que passam pela porta 8081, a porta que trafegará Somente os pacotes do cliente e do servidor.

[10] **GlassFish** é um servidor de aplicação *open source* liderado pela Sun Microsystems para a plataforma Java EE.(Fonte: Wikipédia)

Figura 24: Filtro realizado para identificar pacotes do servidor e do cliente *Web Service*.



Fonte: do Autor.

6 ANÁLISE DE RESULTADOS

Para obter os resultados foram feitos 5 testes de cada quantidade de números aleatório que foi gerada, que se inicia em 1.000, 10.000, 100.000, 200.000 1.000.000.

6.1 RESULTADOS DOS PACOTES OBTIDOS

Os testes e a coleta dos resultados foram executados usando o mesmo cenário citado na seção 5.1 e 4.1. Com base nos 5 testes executados e nos resultados coletados que estão apresentados na tabela 1.

Tabela 1, Relação tráfego de pacotes VS números gerados.

Método	Envio	1.000	10.000	100.0000	200.000	1.000.000[11]
WS	1	20	64	521	1039	*
	2	19	65	513	1030	*
	3	20	64	511	1030	*
	4	20	65	526	1030	*
	5	20	65	529	1030	*
	Média	**20**	**65**	**520**	**1032**	*****
Socket	1	17	77	647	1294	6532
	2	14	77	658	1229	6387
	3	15	77	653	1294	6413
	4	17	74	640	1308	6543
	5	17	78	639	1341	6580
	Média	**16**	**77**	**647**	**1293**	**6491**

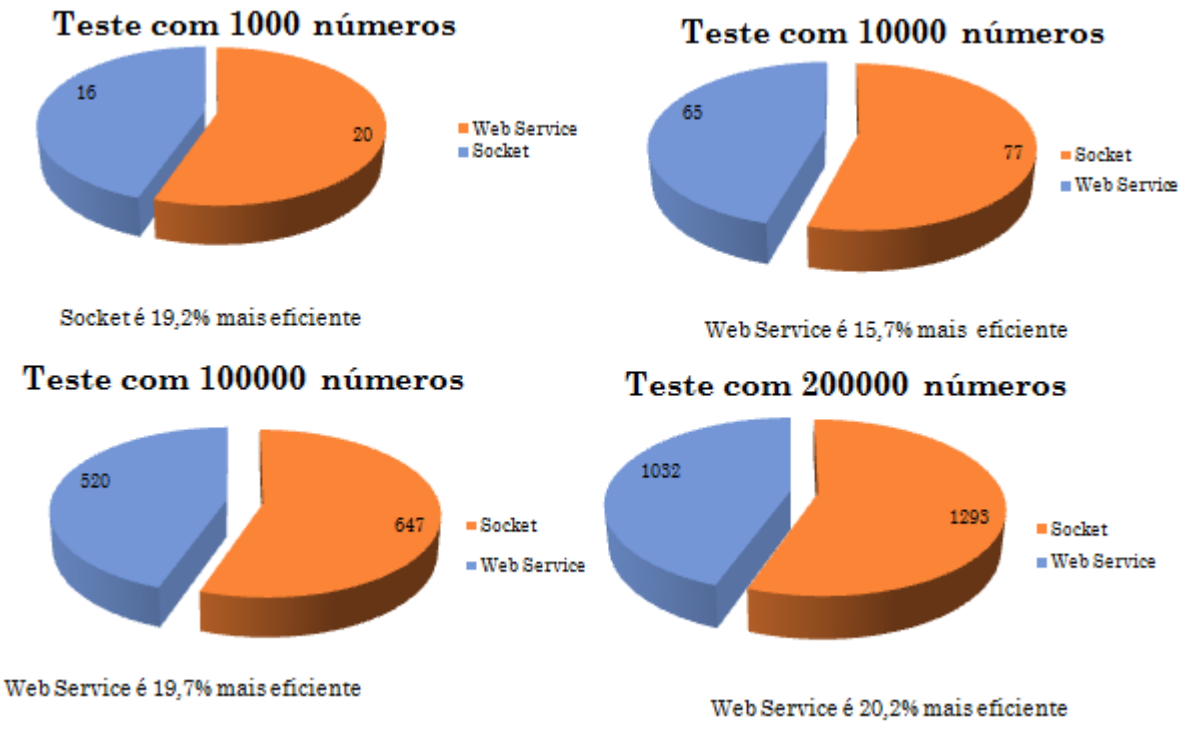
Fonte: do Autor.

- Com 5 transmissões de 1.000 números do cliente para o servidor, a aplicação *Web Service* apresentou uma média de 20 pacotes, enquanto a aplicação feita em *Socket* apresentou uma média de 16 pacotes transmitido na rede.

[11] Nos campos que estão com "*", ocorre por uma limitação que acontece com o *Web Service* que não conseguiu gerar 1 milhão de números, e devido a esse fato foi representado na tabela com os asteriscos.

- Com 5 transmissões de 10.000 números do cliente para o servidor, a aplicação *Web Service* apresentou uma média de 65 pacotes, enquanto a aplicação feita em *Socket* apresentou uma média de 77 pacotes transmitido na rede.
- Com 5 transmissões de 100.000 números do cliente para o servidor, a aplicação *Web Service* apresentou uma média de 520 pacotes, enquanto a aplicação feita em *Socket* apresentou uma média de 647 pacotes transmitido na rede.
- Com 5 transmissões de 200.000 números do cliente para o servidor, a aplicação *Web Service* apresentou uma média de 1032 pacotes, enquanto a aplicação feita em *Socket* apresentou uma média de 1293 pacotes transmitido na rede.
- Com 5 transmissões de 1.000.000 números do cliente para o servidor, a aplicação *Socket* apresentou uma média de 6491 pacotes, em decorrência da limitação do Web Server, mostrado na seção 4.2.2 gerando uma quantidade muito grande de números aleatórios, não foi possível obter os resultados com 1.000.000 de números enviados.

Figura 25: Gráficos representando os resultados dos dados obtidos em relação números de pacotes trafegados a rede



Fonte: do Autor.

6.2 RESULTADOS DE TEMPOS OBTIDOS

A análise do tempo de resposta contém os resultados dos testes realizados no cenário da seção 4.1 e também com o método responsável pela contagem do tempo sendo implementado no cliente, assim calculando o tempo de resposta desde o início do método do cliente até a resposta do servidor. A tabela 2 mostra os dados e a média de pacotes coletados.

Tabela 2, Relação tempo VS números gerados.

Método	Envio	1.000	10.000	100.0000	200.000	1.000.000[12]
WS	1	834	2158	13720	40021	*
	2	537	2381	12797	39939	*
	3	882	2375	11452	39876	*
	4	533	2152	13511	40012	*
	5	454	2364	11146	39734	*
	Média	**648**	**2286**	**12525,2**	**39916,4**	*
Socket	1	592	1093	6002	17489	82988
	2	424	1194	6209	16932	82368
	3	870	1192	4989	18515	88368
	4	874	1685	5864	17935	87466
	5	529	1033	5145	17350	83904
	Média	**657,8**	**1239,4**	**5641,8**	**17644,2**	**85018,8**

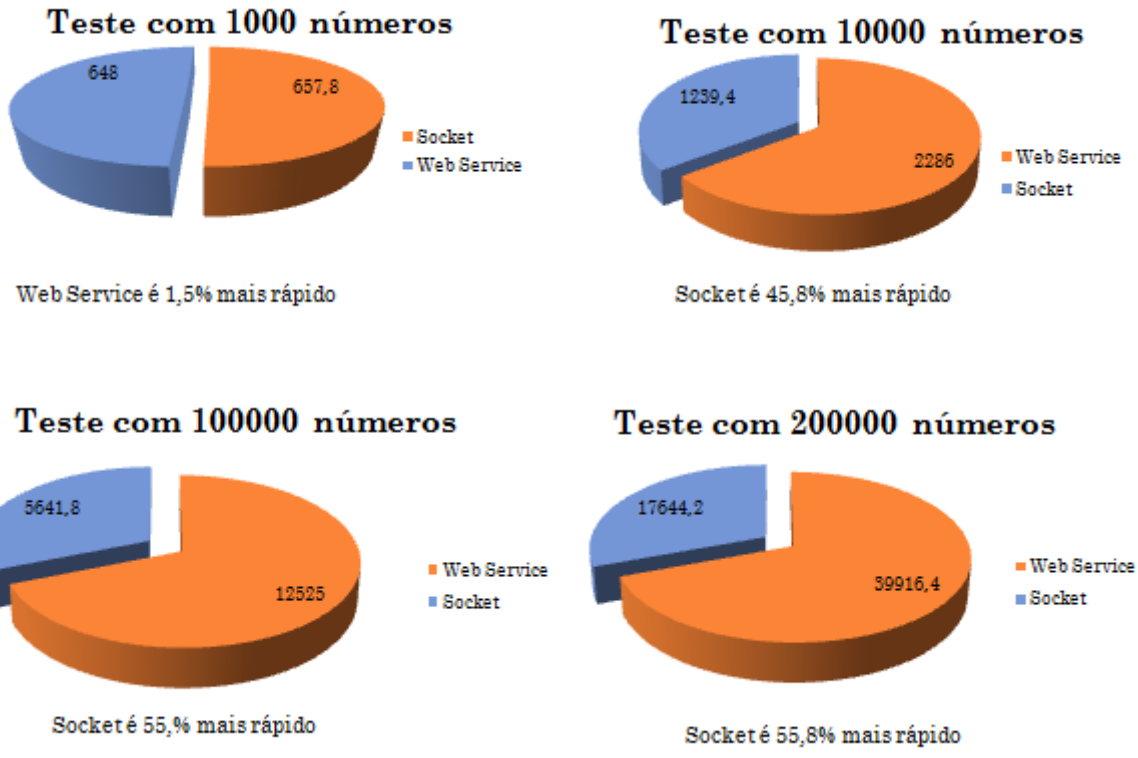
Fonte: do Autor.

- Com 5 transmissões de 1.000 números do cliente para o servidor, a aplicação *Web Service* apresentou uma média de 648 milissegundos, enquanto a aplicação feita em *Socket* apresentou uma média de 657.8 milissegundos, de tempo resposta da aplicação.
- Com 5 transmissões de 10.000 números do cliente para o servidor, a aplicação *Web Service* apresentou uma média de 2286 milissegundos, enquanto a aplicação feita em *Socket* apresentou uma média de 1239.4 milissegundos, de tempo resposta da aplicação.

[12] Nos campos que estão com "*", ocorre por uma limitação que acontece com o *Web Service* que não conseguiu gerar 1 milhão de números, e devido a esse fato foi representado na tabela com os asteriscos.

- Com 5 transmissões de 100.000 números do cliente para o servidor, a aplicação *Web Service* apresentou uma média de 12525.2 milissegundos, enquanto a aplicação feita em *Socket* apresentou uma média de 5641.8 milissegundos, de tempo resposta da aplicação.
- Com 5 transmissões de 200.000 números do cliente para o servidor, a aplicação *Web Service* apresentou uma média de 39916.4 milissegundos, enquanto a aplicação feita em *Socket* apresentou uma média de 17644.2 milissegundos, de tempo resposta da aplicação.
- Com 5 transmissões de 1.000.000 números do cliente para o servidor, a aplicação *Socket* apresentou uma média de 85018.8 milissegundos, enquanto a aplicação feita em *Web Service* não conseguiu suportar 1.000.000 apresentou erro de memória interna.

Figura 26: Gráficos representando os resultados Obtidos em relação ao tempo de resposta.



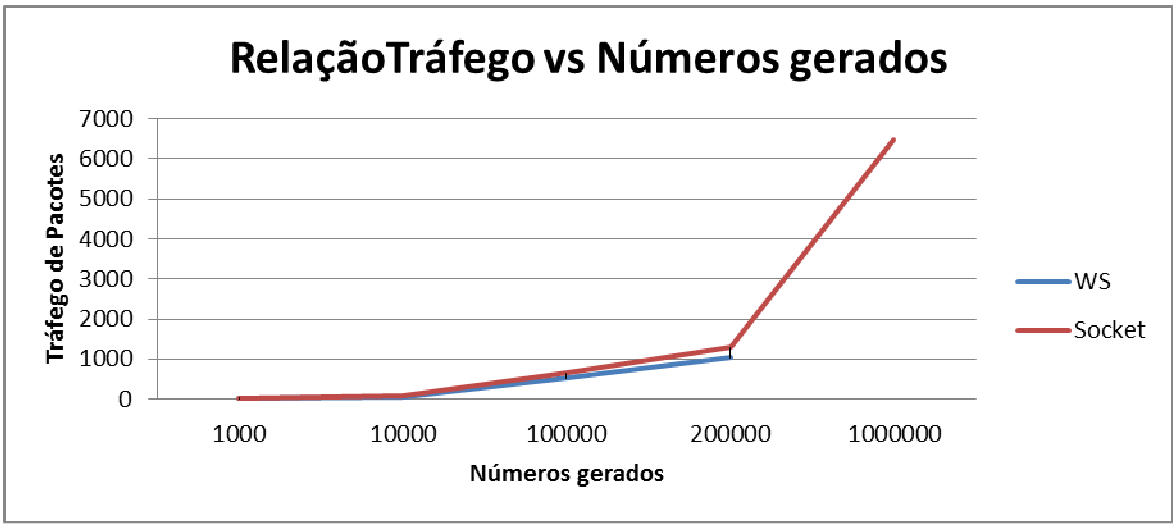
Fonte: do Autor.

A análise dos resultados, obtidos e demonstrados pelas tabelas, está descrito nesta seção com o apoio dos gráficos que seguem abaixo.

6.3 ANÁLISE DO TRÁFEGO DE PACOTES NA REDE

A análise do tráfego de pacotes da rede baseado nos dados coletados e mostrado na tabela1, gerou-se o gráfico 1 para ilustrar os dados.

Gráfico 1, Referente à tabela 1.



Fonte: do Autor.

O gráfico 1 está mostrando na linha azul *Web Service*s, que gerou um número reduzidos de pacotes comparando com o *Socket* que está representado com a linha vermelha.

Pode-se observar que o *Socket*, para o pequeno volume de dados solicitados foi 1.000 de números aleatórios, teve uma ligeira vantagem no tráfego de pacotes comparando com o *Web Services*, que apresentou uma vantagem de 19,2% em relação *Socket.*

Nos testes realizados com 10.000 números aleatórios pode-se notar que o resultado tem uma mudança, pois *Web Services* trafega menos pacotes que o *Socket* quando a

quantidade de números cresceu, sendo então 15,7% mais eficiente na troca de pacotes na rede. Quando o valor de números aleatórios aumenta ainda mais, passando de 10.000 para 100.0000 a vantagem do *Web Services* ainda continua e aumenta para 19,7%.

Em virtude do problema encontrado e mencionado na seção 4.2, foram realizados testes para buscar, dentro do cenário proposto, qual seria o número máximo de números aleatórios que o *Web Services* poderia suportar, com os testes realizados chegou-se em um número máximo que é de 200.000. Para ter o mesmo parâmetro para teste também foi gerado 200.000 em *Socket*, então o resultado foi positivo para o *Web Services* tendo 20,2% de tráfego de dados a menos que o *Socket.*

O *Socket* manda seus números gerados de acordo que os números sejam gerados assim não usando excessivamente o recurso do dispositivo móvel e também conseguindo gerar 1 milhão ou mais números aleatórios, a tabela 1 contém os resultados e a média de pacotes gerados na rede em *Socket*, como *Web Services* não suporta gerar 1 milhão de números, não havendo parâmetro para comparação entre os dois métodos, mas mesmo assim no *Socket* está sendo demonstrado.

6.4 ANÁLISE DE TEMPO DE RESPOSTA

Para a análise do tempo de resposta criou-se o gráfico 2 com base nos dados apresentados na tabela 2.

A análise do tempo de resposta contém os resultados dos testes realizados no cenário da seção 4.1 e também com o método responsável pela contagem do tempo sendo implementado no cliente, assim calculando o tempo de resposta desde o início do método do cliente até a resposta do servidor.

Em uma breve análise na tabela 2 pode-se observar que o *Socket* tem uma vantagem muito pequena no tempo em relação ao *Web Services*. No primeiro teste de 1.000 números o *Web Service* é mais rápido em tempo de resposta, tendo apenas 1,5% de vantagem em relação ao teste realizado com 1.000 números gerados no *Socket*. Em 10.000 transmissões, o *Socket* tem uma boa vantagem em relação ao *Web Services*, ficando 45,8% mais rápido. A diferença é ainda maior quando os testes foram realizados com 100.000, ficando em 55% mais eficiente

que *Web Services*, já com 200.000 a vantagem permaneceu com o *Socket* que ainda é mais rápido em 55,8%.

Gráfico 2, referente à Tabela 2.

Fonte: do Autor.

7 CONSIDERAÇÕES FINAIS

Este trabalho buscou comparar as tecnologias de comunicação *Web Service SOAP* e *Socket stream* que esses métodos são baseados no protocolo de transporte TCP. Dos resultados da comparação houve uma surpresa no trabalho desenvolvido na questão que Web Service consumiu menos pacotes na rede, mas mesmo consumindo menos pacotes ele não conseguiu diminuir o tempo de resposta quando comparado com *Socket.* Enquanto o Socket transmite em torno de 15% a 20% mais pacotes que o Web Service, mesmo assim o Socket consegue ser mais rápido.

Destacamos que a programação Web Service é bem mais simples, devido primeiramente à integração com as interfaces de programações, (Eclipse, NetBeans), que tendo conhecimento inicial da WSDL que é um documento escrito em XML que especifica o serviço e como acessá-lo e quais as operações ou métodos disponíveis. Então já faz a ligação com o serviço e trazendo consigo os métodos disponíveis no servidor de origem da WSDL, assim já estruturando variável que o serviço do servidor precisará para o serviço funcione corretamente. Outra facilidade na programação de Web Service e a facilidade de migrar seu cliente para outra plataforma, sem que o servidor seja modificado.

O *Socket*, por ser um recurso de mais baixo nível que o Web Service, fez-se uma programação que exigiu muito mais para realizar a implementação, e com base em serviços que utilizam Socket quanto maior ou mais complexo o sistema, a programação fica mais difícil, pois para a organização das variáveis transferidas tanto do servidor quanto do cliente, necessitaria de um protocolo que iria fazer essa organização, mas lembrando esse protocolo teria que ser criado manualmente em uma outra classe se possível.

Com este trabalho podemos observar que os dois métodos tem suas características especiais, executando bem a função de transmissão e comunicação entre dispositivos que estão em rede. Sabe-se que na programação precisa-se sair de uma rotina para procurar métodos que resolvam os problemas da melhor forma possível, o método *Web Service* é muito bom para integração de sistemas, por outro lado, é possível usar o *Socket* para se obter um melhor desempenho. Considerando o consumo da rede, observou-se que o método baseado em *Socket* mostrou consumir mais tráfego de rede que o método *Web Service*, contudo essa

diferença não é muito grande, podendo fazer diferença quando em um ambiente que o tráfego de dados é alto.

Por fim, a abordagem deste trabalho permite a abertura de diversos tópicos para serem abordados por trabalhos futuros, tais como:

- Comparar *Socket* com *Web Service* RESTFUL;
- Comparar fazendo uso de outro tipo de aplicação;
- Comparar os métodos usando outro tipo de programação e realizando em outro cenário de teste.

REFERÊNCIAS

ALVES, Otávio, Pedro; Silva S. Robson; TECNOLOGIAS DE SISTEMAS **DISTRIBUÍDOS IMPLEMENTADAS EM JAVA:** SOCKETS, RMI, RMU-IIOP E CORBA. Universidade para o Desenvolvimento do Estado e da Região do Pantanal, 2009.

ABINADER, Abilio; DUEIRE, Rafael; Web Services em Java, 2006,

COMER, Douglas. **INTERLIGACOES DE REDES CIN TCP/IP:** princípios, protocolos e arquitetura. 5ª edição. 2006

COULOURIS, George; DOLLIMORE, Jean; KINDEMBERG, Tim. **SISTEMAS DISTRIBUÍDOS:** conceitos e projeto. 4ª edição. Bookman, 2007.

JAMES F. Kurose; KEITH W. Ross. **REDES DE COMPUTADORES E A INTERNET:** uma abordagem top-down**.** 5ª edição. 2010.

MERGEN,Sérgio; GEYER, Cláudio; RAMOS, M. Emanuel: **WEB SERVICES**: Conceitos, SOAP, WSDL, UDDI. Institude de Informática- UFGRS, Julho, 2008.

MORO, Tharcis D.; DORNELES, Carina F.; REBONATTO, Marcelo T.: **Web services WS-* versus Web Services REST**. REIC - Revista de Iniciação Científica - UFRGS, 2011. Disponível em: <http://seer.ufrgs.br/reic/article/download/22140/12928>. Acesso em: 20/12/2012.

RECKZIEL, Mauricio; **Entendendo os web services.[S.1.],** 2006 Disponivel em: <HTTP://www.imasters.com.br/artigo/4245/webservices/entendendo_os_webservices/>.

SOARES, Luiz Fernando; LEMOS, Guido; COLCHER, Sérgio. **REDES DE COMPUTADORES:** das LANs, MANs, e WANS às redes ATM. 23ª Reimpressão. Elseveir, 1995.

PERPÉTUO, Enio Junior; LOUREÇO, Iverson; Web Services uma Solução para Aplicações Distribuída na Internet, 2003,

MITCHELL, Bradley. **SNIFFERS**, Disponível em: < http://compnetworking.about.com/od/networksecurityprivacy/g/bldef_sniffer.htm>. Acesso em: 18 Dez. 2012.

WEERAWARANA, Sanjiva; CURBERA, Francisco; LEYMANN, Frank; STOREY, Tony; FERGUSON, Donald F. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Adressing, WS-BPEL, WS-Reliable Messaging, and more. Prentice Hall PTR, 2005.

Wikipédia. **SERVIDOR WEB**, Disponível em:<http://pt.wikipedia.org/wiki/Servidor_web >. Acesso em: 15 Mai. 2012.

ANEXOS

ANEXO A: Servidor Socket

```
package br.com.ifsul.tcc;


import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Iterator;
import javax.print.attribute.standard.Média;
import javax.swing.text.html.HTMLEditorKit.Parser;

        public class MyServer {
         public static void main(String[] args){
                média();
         }
         public static long média() {
                 long in = System.currentTimeMillis();

          ServerSocket serverSocket = null;
          Socket socket = null;
          DataInputStream dataInputStream = null;
          DataOutputStream dataOutputStream = null;

          try {
```

```
ServerSocket serverSocket2 = serverSocket = new ServerSocket(6500);
            System.out.println("Listening :6500");

           } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
           }
           while(true){
            try {

socket = serverSocket.accept();

            dataInputStream = new DataInputStream(socket.getInputStream());//cria um socket de entrada.
            dataOutputStream = new DataOutputStream(socket.getOutputStream());//cria um socket de saida.
            ArrayList<Integer> nu= new ArrayList<Integer>();

int q = dataInputStream.readInt();

            for (int i = 0; i < q; i++) {
            nu.add(dataInputStream.readInt());// lendo os números aletórios que vem do cliente.
            System.out.println("vet->"+ nu.get(i));// mostram no console os números criados pelo cliente
            }

            Iterator<Integer> it = nu.iterator();
      double soma = 0;
      while (it.hasNext()) {
         soma += (Integer) it.next();  //realiza a soma dos números que vieram do cliente
      }
```

```
double média = (soma/nu.size()); //realiza a média sobre os números aleatório

System.out.println("Média = " + média);// mostra no console a média

dataOutputStream.writeDouble(média);//mandando para o cliente o resultado da média
aplicada

        } catch (IOException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
        }
        finally{
         if( socket!= null){
          try {
           socket.close();
          } catch (IOException e) {
           // TODO Auto-generated catch block
           e.printStackTrace();
          }
         }

         if( dataInputStream!= null){
          try {
           dataInputStream.close();
          } catch (IOException e) {
           // TODO Auto-generated catch block
           e.printStackTrace();
          }
         }

         if( dataOutputStream!= null){
```

```
    try {
     dataOutputStream.close();
    } catch (IOException e) {
     // TODO Auto-generated catch block
     e.printStackTrace();
    }
   }
  }
 }
}
```

ANEXO B : Cliente Socket

```
package com.example.android.accelerometerplay;


import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Random;

import com.example.android.google.apis.R;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class AndroidClient extends Activity {

EditText textOut, quant;
EditText textOut2;
TextView textIn;

 /** Called when the activity is first created. */
 @Override
```

```
 public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);


    quant = (EditText)findViewById(R.id.edtQuant);
    Button buttonSend = (Button)findViewById(R.id.send);
    textIn = (TextView)findViewById(R.id.textin);
    buttonSend.setOnClickListener(buttonSendOnClickListener);
 }

Button.OnClickListener buttonSendOnClickListener = new Button.OnClickListener(){
public void onClick(View arg0) {


 média();

}

public long média() {
long in = System.currentTimeMillis();

Socket socket = null;
DataOutputStream dataOutputStream = null;
DataInputStream dataInputStream = null;

try {

socket = new Socket("192.168.0.6", 6500);//Criando socket para referenciar o servidor
IP/PORTA
```

```
 dataOutputStream = new DataOutputStream(socket.getOutputStream());//cria um socket de saida.
 dataInputStream = new DataInputStream(socket.getInputStream());//cria um socket de entrada.


 int quanti = Integer.parseInt(quant.getText().toString());//recebendo do usuário quantos números aleatório ele deseja

 ArrayList<Integer> num = new ArrayList();//criando um ArrayList num.

 dataOutputStream.writeInt(quanti);//Escrevendo ou mandando para o servidor a quantia de números aleatório que sera gerada

 for (int i = 0; i < quanti; i++) {
    Random r = new Random();// função gerar números aleatórios
    num.add(r.nextInt(1000));// seta o número do intervalo que será gerado  e o intervalo vai de 0 a 999
    System.out.println("->"+ num.get(i) );//mosta na console os números gerados
    dataOutputStream.writeInt(num.get(i));// está mandando para o servidor cada número aleatório gerado, sendo enviado 1 de cada vez.
    }
 DecimalFormat decimal = new DecimalFormat( "0.00" );//criando um objeto para formatar o resultado com somente 2 números depois da virgula.
 textIn.setText(String.valueOf(decimal.format(dataInputStream.readDouble())));//le o resultado do servidor e mostra no visor do dispositivo móvel.
} catch (UnknownHostException e) {
 // TODO Auto-generated catch block
 e.printStackTrace();
} catch (IOException e) {
```

```
 // TODO Auto-generated catch block
 e.printStackTrace();
}
finally{
 if (socket != null){
  try {
   socket.close();
  } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }

 if (dataOutputStream != null){
  try {
   dataOutputStream.close();
  } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }

 if (dataInputStream != null){
  try {
   dataInputStream.close();
  } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }
}
```

```
 String oo = String.valueOf(System.currentTimeMillis() - in);
 Log.d("Tempo decorrido foi= ", oo);
return System.currentTimeMillis() - in;

  }
 };
}
```

ANEXO C: Servidor Web Service

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com.ifsul.tcc;

import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.Iterator;
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

/**
 *
 * @author InspironZ
 */
@WebService(serviceName = "WS")
public class WS {

    /**
     * Operação de Web service
     */
    @WebMethod(operationName = "média")
    public double média(@WebParam(name = "v1") int v1, @WebParam(name = "v2")
ArrayList v2) {

        int q = v1;
```

```
String aux = v2.get(0).toString();
aux = aux.replace("[", "");
aux = aux.replace("]", "");
String[] a = aux.split(",");                    //adiciona os valores no array "a"

      ArrayList<Integer> nu = new ArrayList<Integer>();

for (String exibe : a) {                        //esse for ta mostrando so os valores que chegaram
do cliente
System.out.println("Vet->" + exibe.trim() + "\n");
nu.add(Integer.parseInt(exibe.trim()));         //está adicionando os valores do cliente no
array "nu"
}
DecimalFormat decimal = new DecimalFormat("0.00");

Iterator<Integer> it = nu.iterator();
Double soma = 0.0;
while (it.hasNext()) {                  //soma
soma += it.next();                      //realiza a soma
}

      System.out.println("Média = " + (soma / q));
      System.out.println("Média = " + decimal.format(soma / q));

return Double.parseDouble(decimal.format(soma / nu.size()).replace(",", "."));

//realiza a média e retorna para o cliente
   }

}
```

ANEXO D : Cliente Web Service

```
package com.WS.ClientWS;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;

import org.ksoap2.SoapEnvelope;
import org.ksoap2.SoapFault;
import org.ksoap2.serialization.SoapObject;
import org.ksoap2.serialization.SoapPrimitive;
import org.ksoap2.serialization.SoapSerializationEnvelope;
import org.ksoap2.transport.AndroidHttpTransport;
import org.xmlpull.v1.XmlPullParserException;

import com.example.android.google.apis.R;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;

public class ClientWS extends Activity {

        //private AndroidHttpTransport transporte;
```

```
    private String resultado;
    private String NAMESPACE = "http://tcc.ifsul.com/";
    private String METHOD_NAME = "média";
    private String SOAP_ACTION = "http://tcc.ifsul.com/WS/médiaReques";
    private String URL = "http://192.168.0.6:8081/ServerWS/WS?WSDL";
    private SoapSerializationEnvelope envelope;



    TextView txtResultado;
    EditText Valor1, Valor2;
    Button btnEnvia;
    Double média;

    @Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);

  Valor1 = (EditText) findViewById(R.id.edtValor1);
  txtResultado = (TextView) findViewById(R.id.txtResultado);
  btnEnvia = (Button) findViewById(R.id.btnEnvia);

  btnEnvia.setOnClickListener(new View.OnClickListener() {

                public void onClick(View v) {
                        // TODO Auto-generated method stub

                        média();
                }
```

```
public long média() {
        long in = System.currentTimeMillis();

        SoapObject request = new SoapObject(NAMESPACE,
METHOD_NAME);


        //Log.d("MARCADOR WS", "Criado e adicionando serviço");

        int quanti = Integer.parseInt(Valor1.getText().toString());

        ArrayList<Integer> num = new ArrayList();

request.addProperty("v1",quanti);



for (int i = 0; i < quanti; i++) {
Random r = new Random();
num.add(r.nextInt(1000));
Log.d("marcador", num.get(i).toString() );
}


request.addProperty("v2", num.toString());//enviando o array inteiro para o servidor


        SoapSerializationEnvelope envelope = new
SoapSerializationEnvelope(SoapEnvelope.VER11);
```

```
                envelope.setOutputSoapObject(request);
            AndroidHttpTransport transporte = new AndroidHttpTransport(URL);
                //Log.d("MARCADOR WS", "Iniciando transporte");
                try {
                    transporte.call(SOAP_ACTION, envelope);
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (XmlPullParserException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                SoapPrimitive response;
                try {
                    response = (SoapPrimitive) envelope.getResponse();
                    resultado = response.toString();
                    Log.d("MARCADOR WS", "Resultado: " + resultado);
                    txtResultado.setText(resultado);
                } catch (SoapFault e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                 String oo = String.valueOf(System.currentTimeMillis() - in);
                 Log.d("Tempo decorrido foi= ", oo);
                return System.currentTimeMillis() - in;
            }
        });

    }

}
```