UNIVERSIDADE FEDERAL DO CEARÁ

SAMUEL PEREIRA DE SOUZA

ANÁLISE DE DESEMPENHO DE APIS DE *SOCKETS*

QUIXADÁ - CEARÁ

2016

SAMUEL PEREIRA DE SOUZA

ANÁLISE DE DESEMPENHO DE APIS DE *SOCKETS*

Monografia apresentada no Curso de Graduação em Redes de Computadores do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial para obtenção do grau de Tecnólogo em Redes de Computadores.

Orientador: Prof. Dr. Arthur de Castro Callado

QUIXADÁ - CEARÁ
2016

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S235a Souza, Samuel Pereira de.
Análise de Desempenho de APIs de Sockets / Samuel Pereira de Souza. – 2016.
43 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Redes de Computadores, Quixadá, 2016.
Orientação: Prof. Dr. Arthur de Castro Callado.

1. Interface de programas aplicativos (Software). 2. Sistema operacionais distribuídos (Computadores). 3. Software - Desenvolvimento. I. Título.

CDD 004.6

SAMUEL PEREIRA DE SOUZA

ANÁLISE DE DESEMPENHO DE APIS DE *SOCKETS*

Monografia apresentada no Curso de Graduação em Redes de Computadores do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial para obtenção do grau de Tecnólogo em Redes de Computadores.

Orientador: Prof. Dr. Arthur de Castro Callado

Aprovada em: __/__/____

BANCA EXAMINADORA

_____

Prof. Dr. Arthur de Castro Callado

Universidade Federal do Ceará - UFC

Orientador

_____

Prof. Me. Antonio Rafael Braga

Universidade Federal do Ceará - UFC

_____

Prof. Me. Allison Barbosa de Souza

Universidade Federal do Ceará - UFC

À meus pais, Guilherme Francisco de Sousa e Francisca Pereira da Silva.

AGRADECIMENTOS

Ao professor Arthur de Castro Callado pela orientação e incentivo.

Aos professores que tive durante este curso de Redes de computadores da UFC, por sua dedicação, comprometimento e compreensão.

Aos familiares, minha namorada e amigos que contribuíram na realização dessa conquista.

“Não importa quanto a vida possa ser ruim, sempre existe algo que você pode fazer, e triunfar. Enquanto há vida, há esperança.” (Stephen Hawking).

RESUMO

A *Internet* mudou a forma como nos comunicamos. Sua revolução levou à criação dos mais variados tipos de sistemas, tendo como proposta facilitar nossas vidas. Esses sistemas então passaram a ter mais exigências quanto a sua qualidade, pois eles devem ser sistemas robustos e confiáveis. Duas das qualidades indispensáveis aos sistemas atualmente são a rapidez e sua escalabilidade: um sistema deve ser rápido o suficiente e escalável. A solução encontrada para esse problema é a distribuição das partes desse sistemas. O que garante essa distribuição é a comunicação em uma rede de computadores. As partes do sistema comunicam-se através da rede usando *sockets,* uma abstração do sistema operacional, que fica abaixo da camada de aplicação, usada para facilitar o desenvolvimento de *softwares* que conversam através da rede. Logo, o *socket* torna-se personagem importante nos requisitos desses sistemas. O objetivo deste trabalho é identificar qual das implementações de *sockets* tem melhor desempenho em que cenários. Para que seja possível uma análise mais profunda na escolha da API (*Application program interface*), este trabalho realizou uma avaliação de desempenho com três APIs de *Sockets*: *ZeroMQ*, *NanoMSG* e *Berkley Sockets*.

Palavras-chave: Sistemas distribuídos, *Sockets*.

ABSTRACT

The Internet has changed the way we communicate. Its revolution led to the creation of the most varied types of systems, with the purpose of making our lives easier. These systems are now having more demands on their quality, as they must be robust and reliable systems. One of the most important qualities of systems today is speed and scalability. A system should be fast enough and scalable. The solution found for this problem is the distribution of the parts of this system. What guarantees this distribution is the communication in a network of computers. The parts of the system communicate over the network using sockets, an abstraction of the operating system, that lies below the application layer, used to facilitate the development of software that talks through the network. Therefore, the socket becomes an important character in the requirements of these systems. The objective of this work is to identify which of the implementations of sockets has better performance, so that a deeper analysis is possible in the choice of API (Application program interface), which as previously mentioned plays an important role in the requirements of distributed systems.

Keywords: Distributed Systems, Sockets.

LISTA DE FIGURAS

LISTA DE GRÁFICOS

LISTA DE TABELAS

LISTA DE ABREVIATURAS: SIGLAS, SÍMBOLOS E ACRÔNIMOS

API - *Application program interface*

CPU - *Central Processing Unit*

DB - *Database*

DNS - *Domain Name Server*

GB - *Gigabyte*

GHz - *Gigahertz*

INPROC - *Inter-Process Communication*

IPC - *Instructions per cycle*

PDF - *Portable Document Format*

POSIX - *Portable Operating System Interface*

RAM - *Random Access Memory*

REST - *Representational State Transfer*

SD - Sistema Distribuído

SO - Sistema Operacional

SOA - *Service-Oriented Architecture*

TCP - *Transmission Control Protocol*

UDP - *User Datagram Protocol*

UFC - Universidade Federal do Ceará

URL - *Uniform Resource Locator*

ZMQ - *ZeroMQ*

ØMQ - *ZeroMQ*

SUMÁRIO

1 INTRODUÇÃO

Com a popularização da Internet, o número de aplicações que trocam informações aumentou vertiginosamente. Graças a essa gigantesca rede, usuários podem desfrutar de sistemas de compras online, redes sociais, sistemas de tempo real entre outros. Sistemas esses que resultam em desafios, mais precisamente na troca de dados a todo momento com requerimentos como: segurança, consistência e confiabilidade (ESTRADA; ASTUDILLO, 2015). A Internet também ajudou na popularização de aplicativos de *streams* que usam a arquitetura "cliente/servidor" para reproduzir canções, vídeos entre outros. Essas aplicações compartilham os mesmos requerimentos das aplicações citadas anteriormente, pois utilizam as mesmas soluções para comunicação.

À medida que a popularidade desses sistemas cresceu, veio a ocorrer a necessidade de sistemas mais robustos e confiáveis. Uma solução que vem atendendo as necessidades são os sistemas distribuídos, que são sistemas que estão ligados uns aos outros por uma rede de computadores (KANGASHARJU, 2008). Essa solução tem sido usada porque proporciona as seguintes características: tolerância a falhas, maior poder de computação para resolver determinado problema, confiabilidade e escalabilidade.

Identifica-se como ponto crítico quanto ao desempenho dos sistemas citados acima a comunicação entre as diversas plataformas existentes e os sistemas. A implementação do mesmo e a escolha certa da API (*Application program interface*) tem papel fundamental no desempenho dos sistemas.

Um ponto identificado em (GRANADOS; RODRIGUEZ; VIVEROS, 2014) é a necessidade de avaliar qual a melhor API para a implementação da solução. Outro ponto necessário para uma avaliação mais elaborada é que tipo de troca de mensagens entre os componentes da solução seria mais adequada. A solução apresentada, no qual este trabalho se baseia, não demonstrou se esses pontos afetam ou não a solução.

O objetivo deste trabalho é mostrar o desempenho das APIs de *sockets*. Serão feitos experimentos para demonstrar qual *socket* API tem melhor desempenho em um determinado cenário de acordo com métricas definidas no processo de desenvolvimento do mesmo.

No capítulo seguinte é apresentada a fundamentação teórica deste trabalho, incluindo definições de sistemas distribuídos, *sockets, ZeroMQ* e *NanoMSG*. Na sequência

são apresentados trabalhos relacionados a este. Em seguida serão mostrados a metodologia proposta, os resultados, as conclusões e trabalhos futuros.

1.1 **Objetivo geral**

O objetivo deste trabalho é mostrar de acordo com medições e repetições de experimentos, qual API de socket tem melhor desempenho na programação de aplicações distribuídas.

1.2 **Objetivos específicos**

- Identificar os modos de programação de cada API estudada;
- Realizar uma avaliação de desempenho das APIs de *socket*;
- Identificar qual *socket* API tem melhor desempenho de acordo com as métricas escolhidas;

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, serão explicados os conceitos de sistemas distribuídos e suas características, o que são *sockets,* sua variação como um sistema distribuído e as API *ZeroMQ, NanoMSG* e *Berkeley Socket*.

2.1 Sistemas distribuídos

Há várias definições de Sistemas distribuídos, todas as quais levam à mesma conclusão. Nesse trabalho foi escolhida a definição de Tanenbaum. “Um sistema distribuído é um grupo de computadores independentes que se mostra aos usuários como um único sistema” (TANENBAUM; VAN STEEN, 2014). A definição implica em algumas características desses sistemas que levam ao dilema do desenvolvimento, que é fazer esses componentes se comunicarem e fazerem parecer ao usuário que o sistema não contém partes independentes. Um exemplo de sistema distribuído é a *Internet*, onde milhares de computadores trocam informações demonstrando ser apenas um único sistema.

Os sistemas distribuídos possuem as seguintes metas: acesso a recursos, transparência da distribuição, abertura e escalabilidade. O acesso a recursos é a meta mais importante desses sistemas, pois toda e qualquer informação do sistema deve ser acessível de maneira controlada e eficiente pelos usuários (TANENBAUM; VAN STEEN, 2014). Recursos podem variar desde arquivos PDF (*Portable Document Format*) para download ao uso de impressoras em rede. Quanto à transparência de distribuição, os usuários devem acreditar que tudo não passa de um sistema. Por exemplo, para os usuários da Internet, tudo está lá. Eles não pensam que cada recurso ou processo está distribuído. Para eles só é preciso estar conectado à rede (*Internet*), para ter acesso ao mundo. Para ser mais detalhado, há alguns tipos de transparência nos SDs (Sistema Distribuído).

Um exemplo, seria a transparência no acesso onde não interessa ao usuário apenas a informação representada. Tomando como exemplo o modelo cliente/servidor, um cliente pode estar usando um navegador específico em um sistema operacional (SO) específico e o servidor rodando em um sistema operacional que pode ser diferente. Para o usuário esses

detalhes não importam, e sim que os dados que cheguem a ele. Caso ele quisesse acessar em outra plataforma, com navegador e SO diferentes, os dados deveriam ser os mesmos.

Outro exemplo é a transparência de localização. Essa diz respeito à forma como o usuário não vê que recursos estão em lugares diferentes ou, mais precisamente, em computadores diferentes. Por exemplo, ao acessar o Facebook, o usuário pode ter a impressão de que está tudo no mesmo lugar. Porém, pode ser que o banco de dados usado para efetuar o login esteja em um servidor diferente do que armazena as postagens do usuário, ou que as imagens fiquem em servidores diferentes dos vídeos. Tudo isso deve ser transparente ao usuário.

A transparência de migração é o modo como um recurso pode mudar de lugar e não ser percebido. Por exemplo, o servidor de um sistema de controle acadêmico, que gerencia as frequências de alunos em uma universidade, fica em um determinado servidor. Esse sistema poderia ser movido para outro servidor e os usuários continuariam acessando o serviço com o mesmo endereço URL (*Uniform Resource Locator*), graças à transparência provida pelo DNS (*Domain Name Server*).

Já a transparência de relocação é similar a anterior, com a diferença que os recursos estão sendo usados no momento da realocação. Ela implica em várias cópias do recurso. Por exemplo, tenho servidores de DB (*Database*) em vários lugares. Se caso, eu viesse a precisar, teria o mesmo recurso disponível imediatamente.

Outra transparência é a de concorrência, um recurso deve ser compartilhado por vários usuários ao mesmo tempo. Um bom exemplo é o Google Docs. Um arquivo pode ser editado ao mesmo tempo por múltiplos usuários. E por último, a transparência de falha. Nessa, deve ser transparente ao usuário que o sistema se recuperou de uma falha.

Continuando com as metas dos sistemas distribuídos, a abertura do sistema é muito importante, no que diz respeito às regras e formas de acesso aos recursos de um sistema. Por exemplo, em uma API REST (*Representational State Transfer*), há caminhos dos recursos que estão disponíveis e as regras para acessá-las, como permissão. Outro fator importante é a interoperabilidade, na qual partes do sistema podem ser escritas com tecnologias diferentes e usando ferramentas diferentes, mas elas devem comunicar-se perfeitamente. Usando o exemplo anterior, através da REST, tanto dispositivos *mobile* quando *desktops* e *mainframes* podem usar os recursos disponíveis, independente da linguagem ou tecnologia usada para criá-los.

A escalabilidade de sistemas distribuídos é a capacidade de crescer e ser eficiente. Essa meta deve ser levada sempre em consideração na projeção de sistemas distribuídos. Por exemplo, um determinado sistema será totalmente escalável se conseguir adicionar tantos usuários quanto o necessário, tantos recursos quanto precisar, ter componentes distribuídos em outros lugares e ainda ser administrável.

Em geral, os sistemas distribuídos podem ser divididos em arquiteturas centralizadas, descentralizadas e híbridas. No modelo cliente/servidor, o servidor fornece o serviço e o cliente o requisita. Já no modelo descentralizado, temos as redes ponto-a-ponto (*peer-to-peer*) estruturadas e não estruturadas. As redes ponto-a-ponto estruturadas possuem um protocolo que garante que cada nó na rede faça um roteamento eficiente na busca de qualquer recurso. Por outro lado, as redes ponto-a-ponto não estruturadas distribuem os nós de forma arbitrária. A híbrida constitui a combinação de uma arquitetura centralizada com uma descentralizada, comumente usadas em sistemas distribuídos que precisam de uma parte centralizada para realizar autenticação de usuário.

Um componente importante dos sistemas distribuídos é o *middleware,* que tem como objetivo resolver o problema da heterogeneidade nos sistemas distribuídos. Segundo COULOURIS et al. (2013), "o *middleware* fornece um modelo computacional uniforme para ser usado por programadores de serviços e aplicativos distribuídos". A comunicação entre os módulos de um sistema distribuído é feita através da rede, no qual dois protocolos da camada de transporte podem usados: o TCP (*Transmission Control Protocol*) e o UDP (*User Datagram Protocol*). É aqui que o *middleware* exerce seu papel fundamental, efetuando a mediação da comunicação entre os diferentes processos.

A comunicação entre os processos é feita através através de datagrama, UDP, ou fluxo TCP (*TCP Stream*). O cliente envia uma requisição para o servidor e aguarda uma resposta. Essa resposta pode ser uma confirmação de operação ou um recurso solicitado. Essa comunicação pode ser de duas naturezas: síncrona e assíncrona. Em uma comunicação síncrona, a requisição e a resposta estão sincronizadas, ou seja, quem fez a solicitação fica em estado de bloqueio até a resposta chegar. COULOURIS et al. (2013) chama as operações de requisitar e responder de operações bloqueantes. Na comunicação assíncrona o tratamento é diferente. O processo requerente não fica bloqueado esperando a resposta. As comunicações tanto UDP quanto TCP usam *sockets,* onde um processo cliente encaminha uma requisição

através do seu *socket* para o *socket* do processo servidor. O UDP usa comunicação assíncrona e o TCP síncrona.

2.2 *Sockets*

Sockets são usados como pontos finais para comunicação entre processos em uma rede de computadores (ORACLE, 2016). Eles garantem comunicação bidirecional entre processos, abstraindo complexidades do meio de transmissão entre a camada de aplicação e a de transporte. Para uma análise mais completa sobre *sockets* consultar (STANISLAV, 2016) e (GROUP, 2016).

Sockets tem um ciclo de vida dividido em quatro partes. A primeira inclui a criação e a destruição do socket. A segunda é a configuração, onde opções são passadas e verificadas. A terceira parte é a criação das conexões. A quarta, o uso para escrita (envio de dados) e leitura (recepção de dados). Há diversas implementações deles, como *ZeroMQ* (ØMQ, 2016), *NanoMSG* (ANTUKH, 2016), *Berkeley-sockets (*MILLER, 2016) *e Apache-mina* (FOUNDATION, 2016).

2.3 *ZeroMQ*

O *ZeroMQ* é uma API *socket* projetada para facilitar o desenvolvimento de aplicações distribuídas. Uma definição mais formal pode ser encontrada em (ØMQ, 2016). Essa API foi concebida para ter o mínimo de complexidade (ØMQ, 2016), facilitando o desenvolvimento.

Para desenvolver uma aplicação com essa API são necessários três pequenos passos, segundo PIËL (2016). Primeiro deve-se escolher a tecnologia de transporte, depois é necessário escolher a infraestrutura e por último, escolher qual padrão de mensagem será usado.

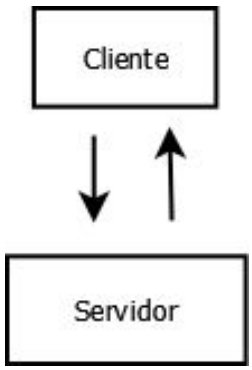
O *ZeroMQ* oferece aos desenvolvedores quatro tecnologias para transporte. O TCP, *MULTICAST*, IPC (*Instructions per cycle*) e INPROC (*Inter-Process Communication*).

A escolha da infraestrutura é necessária, pois muitas vezes é necessário um intermediário na rede para lidar com as pontas. Três dispositivos são disponibilizados pelo *ZeroMQ*: *queuer, forwarder e streamer*. O *queuer* é utilizado para o padrão *Request-Reply*. O *forwarder* é usado no pub-sub, e por último o *streamer* é usado pelo *pipeline*.

A API disponibiliza quatro padrões básicos: *Request-reply*, Pub-sub, *Pipeline* e o *Exclusive pair*. O primeiro é o tradicional modelo de requisição-resposta, onde um cliente envia uma requisição e o servidor responde.

A figura 1 mostra a esquematização desse padrão, onde um cliente envia uma requisição e o servidor envia uma resposta. No pub-sub, um nó envia uma mensagem para vários nós. Uma boa analogia para entender esse padrão é de uma lista de *e-mail*. Um *e-mail* é enviado a todos os assinantes (*subscribers*) da lista.

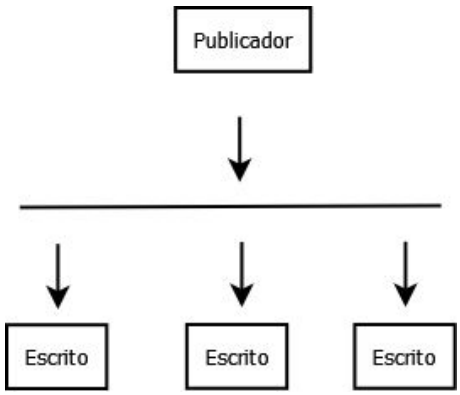
Figura 1. Padrão Request-Reply.



Fonte: (ØMQ, 2016), adaptada pelo autor.

A figura 2 mostra o padrão, no qual um nó envia uma mensagem para três nós inscritos para receber mensagens dele. O padrão *Pipeline* divide uma tarefa entre vários nós (mais de um) e por consequência essas partes podem ser recebidas por um ou mais nós.

Figura 2. Padrão Pub-Sub.

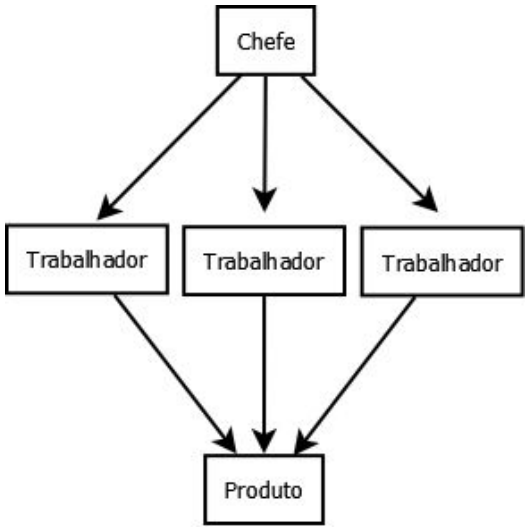


Fonte: (ØMQ, 2016), adaptada pelo autor.

A figura 3 demonstra o funcionamento do *pipeline*, no qual uma tarefa é dividida entre trabalhadores, resultando em um produto final. Por último, o padrão *exclusive pair*

amarra um *socket* a outro. Ele é similar ao primeiro padrão. O que os diferencia é apenas a simplicidade que o *exclusive pair* adota na configuração da comunicação.

Figura 3. Padrão Pipeline.



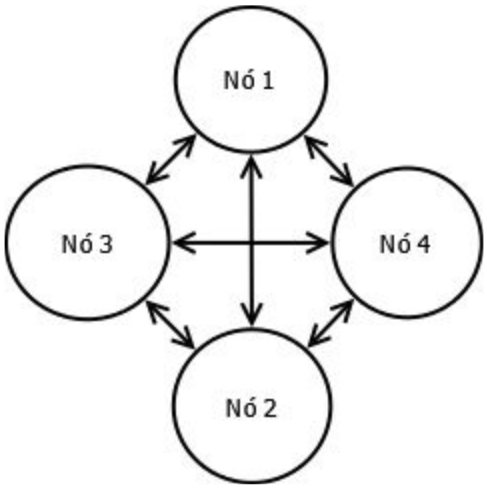
Fonte: (KOSTOV, 2016), adaptada pelo autor.

2.4 **NanoMSG**

A API *NanoMSG* é uma concorrente do *ZeroMQ*. *NanoMSG* é uma biblioteca que provê padrões de comunicação comum (ANTUKH, 2016). Suas principais intenções são: aumentar a rapidez, prover escalabilidade da camada de rede e facilitar seu uso no desenvolvimento. *NanoMSG* provê, além dos padrões que estão disponíveis no *ZeroMQ*, interfaces para o uso de novos protocolos de transporte como *WebSockets*, por exemplo. Outros fatores presentes são a sintaxe e semântica totalmente de acordo com o padrão POSIX (*Portable Operating System Interface*). Os *sockets* em *NanoMSG* além de ter mecanismos internos para a otimização na CPU (*Central Processing Unit*), são *thread-safe*. "Uma função *thread-safe* pode ser invocada simultaneamente por múltiplas *threads,* mesmo quando as invocações compartilham dados, porque todas as referências são serializadas" (LTD., 2016).

Os padrões que o *NanoMSG* oferece são os padrões do *ZeroMQ*, além do *BUS* e *SURVEY*. O *BUS* pode ser resumido como *many-to-many communication.* Em outras palavras, cada nó em um grupo pode enviar mensagens para todos os outros nós. A figura 4 mostra o padrão *BUS*, no qual todos os nós podem enviar mensagens entre si.

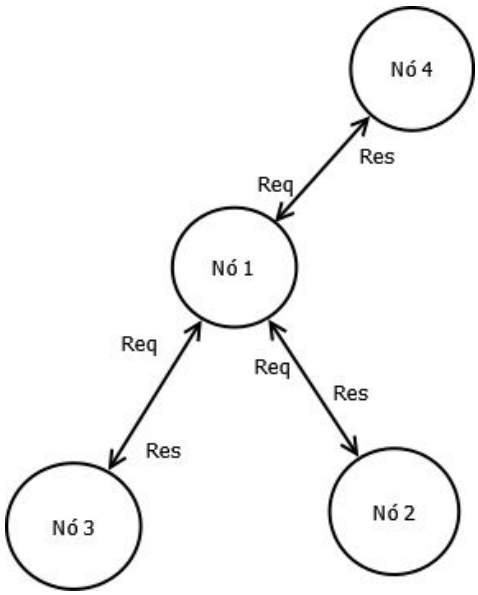
Figura 4. Padrão BUS.



Fonte: (GEEK, 2016), adaptada pelo autor.

O *SURVEY* já é bem visível. Um nó faz uma pergunta a um grupo de nós. Todos os nós respondem ao nó que perguntou. A figura 5 mostra o padrão *SURVEY*, em que um nó pergunta a todos os nós e esses nós respondem.

Figura 5. Padrão Survey.



Fonte: (GEEK, 2016), adaptada pelo autor.

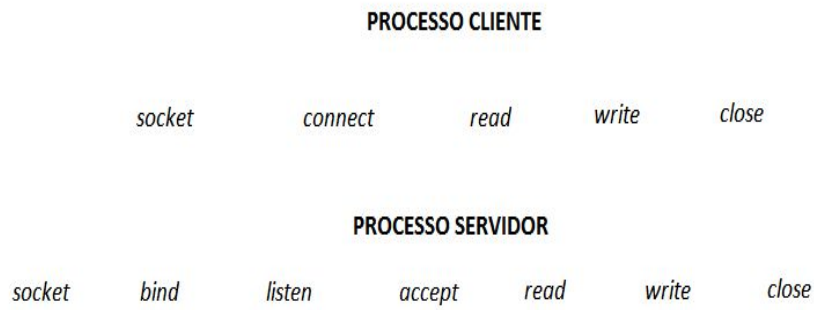
2.4 Berkeley *Sockets*

Segundo Miller (2016), os *Berkeley Sockets* foram desenvolvidos no começo da década de 1980 na *University of California at Berkeley*. Essa API suporta dois tipos de comunicação: UDP e TCP.

As principais chamadas de sistema da API *Berkeley* são: *socket, connect, write, read, close, bind, listen, accept, select*. A chamada *socket* cria um *socket* para a comunicação e retorna uma descrição do socket em arquivo. A *connect,* usado na comunicação com TCP, é usada pelo processo cliente para estabelecer uma conexão com o processo servidor. Já a chamada *write* é usada para enviar os dados na comunicação. Seu oposto é a *read,* que retorna o número de *bytes* recebidos na conexão. A *close* é usada para desalocar o *socket* criado. A chamada *bind,* disponível para os dois tipos de comunicação, é usada pelo processo servidor para associar um processo cliente com o socket. Outra chamada usada pelo servidor é o *listen.* Essa chamada indica que o processo está pronto para receber conexões. A *accept* é usada para aceitar uma conexão e a *select* é usada para determinar se há dados disponíveis no *socket.*

A figura 6 mostra as chamadas de sistema por cada processo.

Figura 6. Chamadas de sistema por cada processo TCP.



Fonte: o próprio autor.

As chamadas na figura acima são utilizadas em comunicação TCP. As chamadas *socket, binde* e *close* também são usadas em comunicação UDP. Para realizar as operações de enviar e receber mensagens, são utilizadas as chamadas *sendto* e *recvfrom.*

3 TRABALHOS RELACIONADOS

Em (GRANADOS; RODRIGUEZ; VIVEROS, 2014), os autores criaram uma solução para lidar com a heterogeneidade de Sistemas Especialistas Distribuídos. A solução pretende promover escalabilidade, tolerância a falhas e alta disponibilidade. Através de uma interface REST, facilitam o acesso de usuários ao sistema através da independência das plataformas que esses mesmos usuários usam. Segundo a descrição da proposta, o *middleware* aceita requisições dos usuários através de uma interface REST, ele interage com o sistema através dos *sockets* da API *ZeroMQ*. As mensagens são enviadas para os nós do sistema através de uma requisição.

A semelhança entre (GRANADOS; RODRIGUEZ; VIVEROS, 2014) é que ambos os trabalhos utilizam sistemas distribuídos. A diferença é que no trabalho proposto a ideia é analisar o desempenho da API que gerencia a comunicação entre as partes do sistema. O trabalho de (GRANADOS; RODRIGUEZ; VIVEROS, 2014) resolve o problema de independência das plataformas do sistema especialista distribuído.

No trabalho de (BILAL; MOHSIN, 2012), o sistema especialista distribuído é desenvolvido usando SOA (*Service-Oriented Architecture*). Seus objetivos são: resolver problemas de DES (*Distributed Expert Systems*) não identificados, fazer o *Muhadith*, como o sistema é chamado, e atuar como um *expert* nas leis do Islamismo.

As diferenças e semelhanças entre o trabalho proposto e (BILAL; MOHSIN, 2012) são as mesmas entre este trabalho e (GRANADOS; RODRIGUEZ; VIVEROS, 2014).

Em (ESTRADA; ASTUDILLO, 2015), os autores comparam duas APIs, *ZeroMQ* e *RabbitMQ*. Seu objetivo é identificar qual das duas tem melhor desempenho, porém avaliando as características intermediárias para envio e recebimento de mensagens.

Este trabalho resolve a necessidade de comparar as tecnologias de implementação dos *middlewares* em (GRANADOS; RODRIGUEZ; VIVEROS, 2014) e (BILAL; MOHSIN, 2012), pois ambos não justificaram a escolha da tecnologia *ZeroMQ* para a implementação. Identificamos que essa escolha causa impacto diretamente no desempenho de tais sistemas por estar ligada ao recebimento e envio de mensagens na rede.

No trabalho (ESTRADA; ASTUDILLO, 2015), os autores comparam uma solução de *socket* em outra camada. Neste caso, a API *ZeroMQ* atuou como um *Messenger Queuer* e é comparada com uma solução que atua somente como *Messenger Queuer.* Este trabalho avalia

o *ZeroMQ* em um nível mais baixo. Ou seja, *ZeroMQ* atuando como a abstração que permite dois processos enviar e receber mensagens.

4 SOLUÇÃO PROPOSTA

Nesta seção serão explicados os procedimentos metodológicos executados no trabalho e a solução para resolver o problema identificado.

4.1 Procedimentos metodológicos

4.1.1 Escolha das APIs

O primeiro passo para a realização do trabalho foi a identificação das APIs disponíveis na comunidade. A escolha das mesmas foi feita levando em consideração os pontos cruciais para o desenvolvimento da solução, que se seguem. Primeiro, as APIs deveriam ter seus respectivos códigos abertos. Segundo, deveriam estar em conformidade com o padrão POSIX. Ela deve ser compatível com o padrão POSIX porque esse padrão é o conjunto de padrões para construção de sistemas operacionais, como linux. Sabemos que grande parte dos sistemas distribuídos são desenvolvidos tendo como alvo esses sistemas operacionais. Outro ponto levado em consideração foi a popularidade das APIs, de forma que a falta de documentação e discussão de problemas não seja uma barreira para o trabalho, uma vez que a resolução de problemas fugiria ao escopo do trabalho. E por último, os protocolos de escalabilidade.

Foram escolhidas três APIs: *Berkeley-socket, ZeroMQ e NanoMSG*. Elas foram escolhidas porque atingiram os requisitos definidos no parágrafo acima e possuírem a mesma linguagem de implementação.

4.1.2 Desenvolvimento dos clientes e servidores

Os códigos usado para os experimentos foram desenvolvidos levando em consideração as APIs escolhidas para avaliar o desempenho. Neste trabalho foi utilizado o modelo cliente/servidor, um dos modelos mais populares atualmente. O código foi baseado nas ferramentas disponibilizadas pelas APIs para avaliação de desempenho, com exceção da API *Berkeley*. Para a API *Berkey* foi usando um código disponível em Ug (2016), adaptado para os experimentos.

A linguagem escolhida foi C, pois todas as APIs foram desenvolvidas inicialmente com essa tecnologia. Está além do escopo deste trabalho, avaliar o desempenho levando em consideração a linguagem de implementação das APIs.

4.1.3 **Escolha das métricas de avaliação**

As métricas escolhidas para avaliação são vazão e latência. A medição da vazão, que é a quantidade de mensagens lidas em um determinado tempo, será realizada no receptor. A latência, que determina o tempo gasto para leitura de cada mensagem, também será medida no receptor.

4.1.4 **Fatores e níveis**

Para o cálculo do tempo de resposta, foram usados os seguintes fatores e níveis, conforme a Tabela 1. O tempo de medida é em segundo, utilizando a lib sys/time.h para calcular o tempo de resposta de cada mensagem recebida. Utilizou-se a duração de 1 hora para o cálculo de tempo de resposta, gerando a cada segundo desse período, uma amostra.

Tabela 1. Fatores e níveis.

Fatores	Níveis
Nº mensagens enviadas	30/s, 50/s
Tamanho das mensagens	10KB, 100KB

Fonte: o próprio autor.

Para o cálculo da vazão, o transmissor não faz qualquer limitação no envio, usando a capacidade máxima possível dadas as limitações de hardware. Utilizou-se a duração de 1 minuto para o cálculo de vazão, com amostra a cada segundo, totalizando 60 amostras.

Para analisar a vazão, utilizamos um único cenário. Uma máquina enviando a quantidade que ela conseguisse enviar por segundo. Cada mensagem tinha o tamanho de 100 KB.

4.1.5 **Cálculo dos intervalos de confiança**

O tempo de duração de cada experimento foi de uma hora, o que resultou em muitas amostras. Foi feita a média de tempo das mensagens para cada segundo, totalizando 3.600 amostras para análise no cálculo de tempo de resposta e 60 amostras no cálculo de vazão. O nível de confiança foi estabelecido em 95%.

4.1.5 C**oleta dos dados e elaboração de gráficos**

Os experimentos rodaram por uma hora, gerando 3600 amostras, uma vez que foi feita a média de cada segundo. Para os experimentos de vazão, reduziu-se os dados coletados para 60 amostras, calculando a média de cada segundo durante um minuto. Os *logs* gerados foram salvos em arquivos de texto. Após o término dos experimentos, os arquivos foram analisados usando *Excel* e os gráficos de comparação foram gerados.

5 RESULTADOS

Nesta seção, serão apresentados os resultados encontrados no trabalho. Foram coletadas entre 108.000 e 180.000 mensagens enviadas para o experimento de tempo de resposta, pois os níveis do fator quantidade de mensagens variaram entre 30 e 50 mensagens por segundo, totalizando uma hora de experimento para cada fator. O tamanho das mensagens serem 10 KB e 100 KB respectivamente deve-se ao fato que esses tamanhos já apresentariam uma variação nos tempos. Outro motivo seria a limitação da máquina, que não conseguiria enviar dados maiores em período de tempo adequado para o trabalho. Esses motivos também se aplicam ao número de mensagens por segundo. Para o experimento de vazão, foram enviadas mensagens de 100 KB vezes a quantidade de emissores.

Para a execução dos experimentos, foram utilizadas duas máquinas virtuais. As duas possuem 2 GB (*Gigabyte*) de memória RAM (*Random Access Memory*), 8 GB de disco rígido. Ambas estão rodando Ubuntu Server 14.04. O *software* de virtualização que está sendo utilizado é o VirtualBox (ORACLE, 2016). A máquina que hospeda essas duas máquinas virtuais é um DELL com processador Intel Core i5 de 2,20 GHz (*Gigahertz*), com 8 GB de memória RAM, Windows 10 64 bits. Foi utilizada uma rede interna no VirtualBox configurada para execução dos experimentos

5.1 Coleta e Análise

A coleta dos dados foi realizada no período de 1o. de Dezembro a 10 de Dezembro de 2016. Os dados gerados foram salvos em arquivos para a análise posterior.

5.1.1 Tempo de Resposta

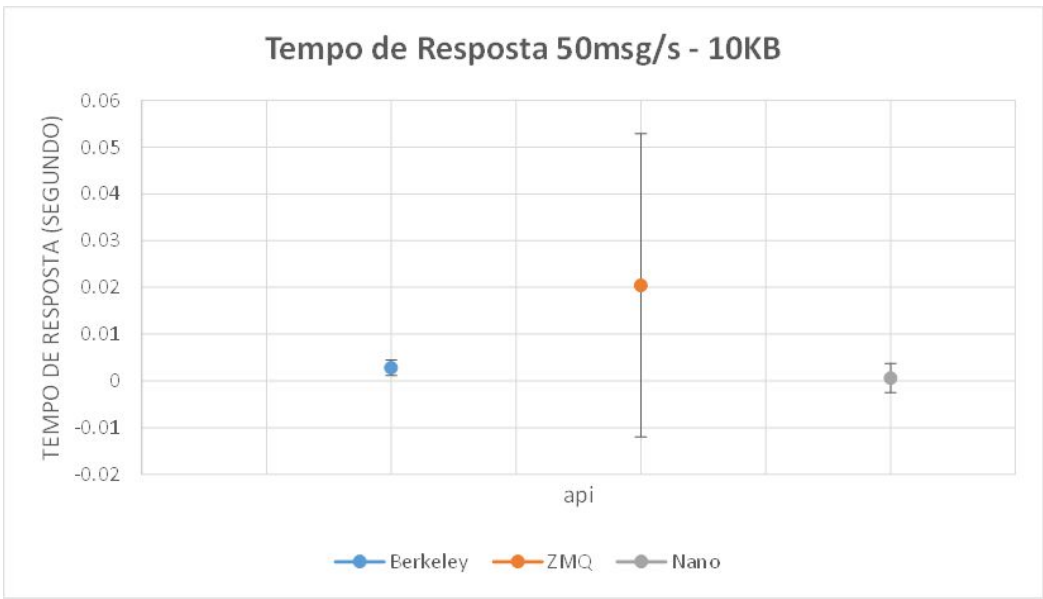
O gráfico 1, logo abaixo, mostra os dados coletados no experimento de tempo de resposta. Neste experimento, uma máquina (cliente) enviou 30 mensagens por segundo, cada mensagem sendo de tamanho 10 KB. É possível observar que das três APIs, a API *NanoMSG* obteve um melhor desempenho em relação às outras opções. O intervalo de confiança nos mostra que o *ZeroMQ* teve uma grande variação no tempo de resposta comparado às outras APIs.

Gráfico 1: Tempo de Resposta 30msg/s - 10 KB.

Fonte: o próprio autor.

O gráfico 2 mostra os resultados de uma máquina (cliente) que enviou 50 mensagens por segundo, cada mensagem sendo de tamanho 10 KB. É possível observar que das três APIs, a API *NanoMSG* novamente obteve um melhor desempenho médio em relação às outras opções.

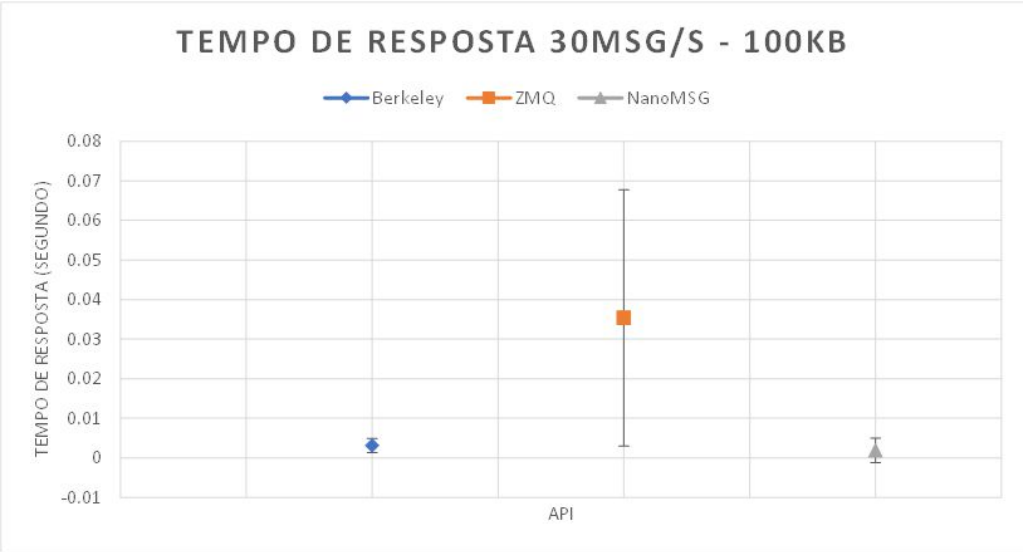
Gráfico 2. Tempo de Resposta 50msg/s - 10 KB.



Fonte: o próprio autor.

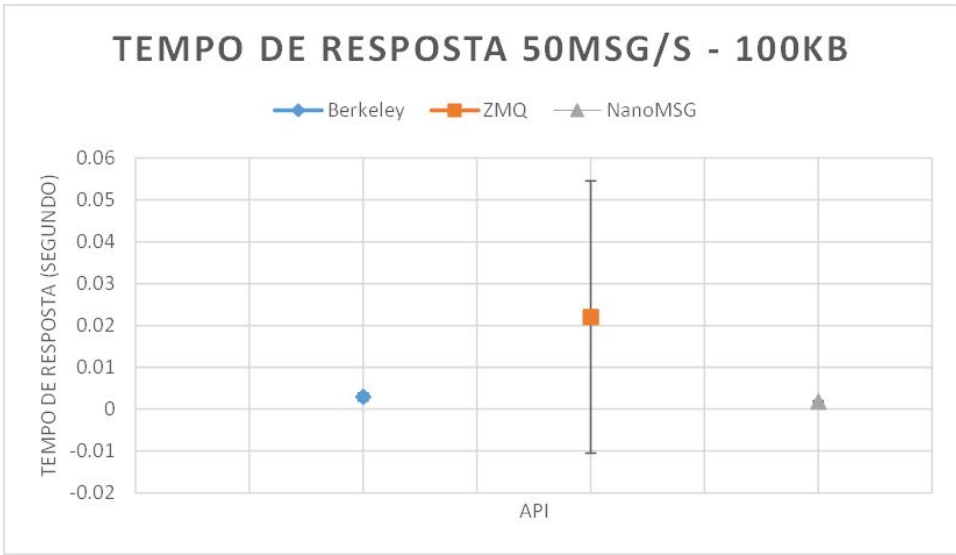
No gráfico 3 é possível observar que equilibrou, porém a API *NanoMSG* continuou com melhor desempenho em relação às outras opções. Esse cenário é configurado com uma máquina enviando novamente 30 mensagens por segundo, porém com 100 KB de tamanho. O intervalo de confiança nos mostra que nesse cenário, as três APIs podem ter o mesmo rendimento, embora a variabilidade da API *ZeroMQ* seja significativamente maior.

Gráfico 3. Tempo de Resposta 30msg/s - 100 KB.



Fonte: o próprio autor.

Gráfico 4. Tempo de Resposta 50msg/s - 100 KB.



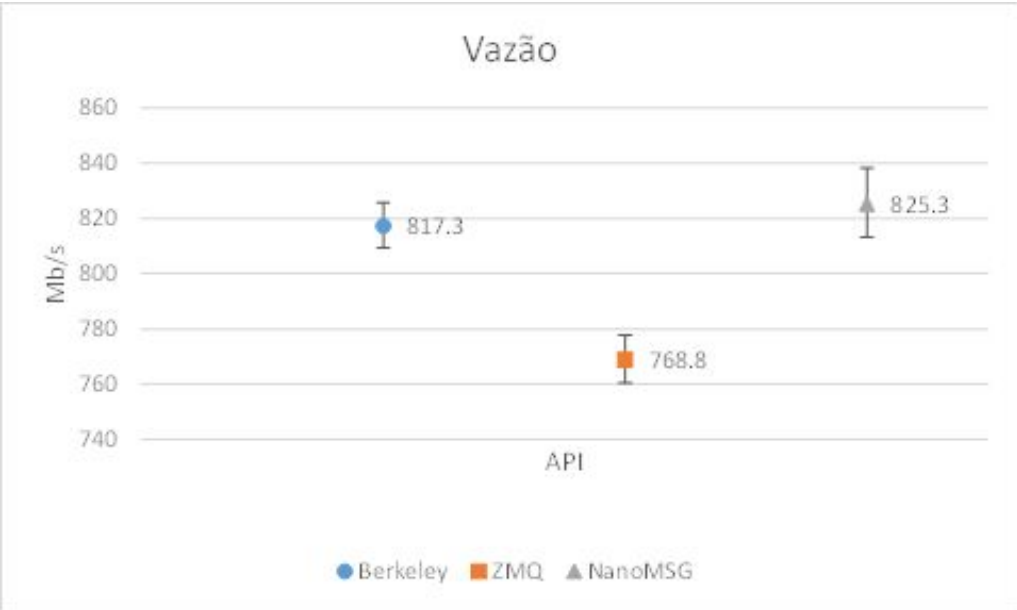
Fonte: o próprio autor.

O gráfico 4 mostra os dados coletados no cenário onde uma máquina envia 50 mensagens por segundo, porém com 100 KB de tamanho. Nesse cenário os resultados foram quase iguais os dos cenários anteriores, diferenciando apenas o intervalo de confiança entre *Berkeley* e *NanoMSG*.

5.1.2 **Vazão**

Para o experimento de vazão, cliente e servidor enviaram dados sem limite de mensagens por segundo. Foram usadas mensagens de 100 KB. Os dados apresentados no gráfico 5 nos mostram o resultado do experimento de vazão. Nele, observamos que novamente a API *NanoMSG* teve um melhor rendimento, mas com desempenho muito parecido com Berkeley.

Gráfico 5. Vazão.



Fonte: o próprio autor.

Detalhe importante sobre os gráficos. Embora os intervalos de confiança tenham passado de zero em alguns casos, isso ocorreu devido a uma dispersão assimétrica, sendo que esses valores são impossíveis de ocorrer na prática.

5.1.2 **Discussão**

A tabela 2 mostra a comparação entre as três APIs avaliadas. O primeiro critério, quantidade de código gerado, a *ZeroMQ* teve um desempenho surpreendente. Apenas nove linhas de código são necessárias para realizar as operações criar socket, conectar, enviar dados, receber dados, e fechar conexão. Isso também resulta na facilidade de programação, pois menos código é gerado, diminuindo a quantidade de erros possíveis e retirando a complexidade na escrita do código. Em contrapartida, *Berkeley* e *NanoMSG* precisam gerar muito mais código do que *ZeroMQ*, mais precisamente 23 e 20 linhas de código respectivamente, no lugar das 9 linhas do *ZeroMQ*.

Tabela 2. Comparação entre APIs

API	**Quantidade de Código Gerado**	**Facilidade de Programação**	**Desempenho**
Berkeley	Muito	Pouco	Excelente
ZeroMQ	Pouco	Muito	Razoável
NanoMSG	Muito	Média	Excelente

Fonte: o próprio autor.

6 CONCLUSÃO

Este trabalho teve como objetivo mostrar qual API *socket* tem melhor desempenho. Os experimentos foram executados no período de 1 de Dezembro a 10 de Dezembro de 2016 no qual um uma máquina, cliente, envia requisições para uma outra máquina, servidora.

A análise utilizou duas métricas: tempo de resposta e vazão. Os dados mostraram que a *NanoMSG* obteve o melhor desempenho para os cenários dos experimentos. Com essa conclusão, o trabalho pode ser usado como referência pela comunidade na escolha de uma dessas opções que atendam suas necessidades, sabendo que apenas uma delas foi melhor para os cenários descritos nas seções anteriores.

6.1 Trabalhos Futuros

Como sugestão para trabalhos futuros, o interessado deverá executar os experimentos em máquinas físicas de alto desempenho para o lado servidor. A aleatoriedade pode ser obtida colocando essas máquinas em uma rede pública, como a dos alunos do campus da UFC (Universidade Federal do Ceará) Quixadá. Por último, com a finalidade de mostrar um cenário mais comum, poderá ser implementado um mini servidor HTTP (GROUP et al., 2016).

Outro trabalho que pode ser abordado no futuro é a comparação entre APIs implementadas em outras linguagens populares, como Java. Também seria interessante avaliar as APIs com SOA.

6.2 Desafios encontrados

No desenvolvimento deste trabalho, houve algumas dificuldades, como falta de recursos de *hardware,* um dos motivos para a execução dos experimentos usando máquinas virtuais. Houve também a falta de aleatoriedade no tráfego de rede, porém a complexidade foi reduzida ao máximo.

REFERÊNCIAS

ANTUKH, Andrey. Jnanomsg documentation. Disponível em: <http://niwinz.github.io/jnanomsg/latest/>. Acesso em: 08 set. 2016.

BILAL, Kashif; MOHSIN, Sajjad. Muhadith: A Cloud Based Distributed Expert System for Classification of Ahadith. 2012 10th International Conference On Frontiers Of Information Technology, [s.l.], p.73-78, dez. 2012. Institute of Electrical & Electronics Engineers (IEEE). http://dx.doi.org/10.1109/fit.2012.22.

COULOURIS, George et al. Sistemas Distribuídos: Conceitos e Projeto. 5. ed. -: Bookman, 2013. Disponível em: <https://books.google.com.br/books?id=6WU3AgAAQBAJ&printsec=frontcover&dq=sistemas+distribuidos+pdf&hl=en&sa=X&ved=0ahUKEwjTyISM-vbOAhUCS5AKHZGICW4Q6AEIHjAA#v=onepage&q=sistemas distribuidos pdf&f=false>. Acesso em: 04 set. 2016.

ESTRADA, Nicolas; ASTUDILLO, Hernan. Comparing scalability of message queue system: ZeroMQ vs RabbitMQ. 2015 Latin American Computing Conference (clei), [s.l.], p.1-6, out. 2015. Institute of Electrical & Electronics Engineers (IEEE). http://dx.doi.org/10.1109/clei.2015.7360036.

FOUNDATION, The Apache Software. Apache MINA. Disponível em: <https://mina.apache.org/>. Acesso em: 17 dez. 2016.

GEEK, Brave New. A Look at Nanomsg and Scalability Protocols (Why ZeroMQ Shouldn't Be Your First Choice). Disponível em: <http://bravenewgeek.com/a-look-at-nanomsg-and-scalability-protocols/>. Acesso em: 13 set. 2016.

GRANADOS, Guillermo Barrera; RODRIGUEZ, Jose; VIVEROS, Amilcar Meneses. Middleware architecture for control an heterogeneous expert system. 2014 11th International Conference On Electrical Engineering, Computing Science And Automatic Control (cce), [s.l.], p.1-6, set. 2014. Institute of Electrical & Electronics Engineers (IEEE). http://dx.doi.org/10.1109/iceee.2014.6978313.

GROUP, Network Working et al. RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1. Disponível em: <https://tools.ietf.org/html/rfc2616>. Acesso em: 07 dez. 2016.

GROUP, Network Working. Request for Comment 147. Disponível em: <https://tools.ietf.org/html/rfc147>. Acesso em: 05 set. 2016.

JABBAR, Haider Khalaf; KHAN, Rafiqul Zaman. Survey on Development of Expert System in the Areas of Medical, Education, Automobile and Agriculture. 2015 2nd International Conference On Computing For Sustainable Global Development (indiacom). New Delhi, p. 776-780. mar. 2015.

KANGASHARJU, Jussi. Distributed Systems: What is a distributed system? 2008. Disponível em: <https://www.cs.helsinki.fi/u/jakangas/Teaching/DistSys/DistSys-08f-1.pdf>. Acesso em: 13 jul. 2016.

LTD., © 2016 The Qt Company. Qt Documentation: Reentrancy and Thread-Safety. Disponível em: <http://doc.qt.io/qt-5/threads-reentrancy.html>. Acesso em: 17 dez. 2016.

ØMQ. ØMQ - The Guide. Disponível em: <http://zguide.zeromq.org/page:all>. Acesso em: 08 set. 2016.

KOSTOV, Dimitar. Dimitar Kostov ramblings. Disponível em: <http://mytrile.github.io/blog/2010/09/20/zeromq-messaging-patterns/>. Acesso em: 13 set. 2016.

MILLER, Vic. Berkeley Socket. Disponível em: <https://phobos.ramapo.edu/~vmiller/AdvancedUnix/berkeley_socket.htm>. Acesso em: 30 nov. 2016.

ORACLE. Welcome to VirtualBox.org! Disponível em: <https://www.virtualbox.org/>. Acesso em: 03 dez. 2016.

ORACLE. What Is a Socket? Disponível em: <https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html>. Acesso em: 05 set. 2016.

PIËL, Nicholas. ZeroMQ An introduction. Disponível em: <http://nichol.as/zeromq-an-introduction>. Acesso em: 08 set. 2016.

STANISLAV., G. Adam. Chapter 7. Sockets: Part II. Interprocess Communication. Disponível em: <https://www.freebsd.org/doc/en/books/developers-handbook/sockets.html#sockets-synopsis>. Acesso em: 05 set. 2016.

TANENBAUM, Andrew S.; VAN STEEN, Maarten. Distributed Systems: Principles and Paradigms. 2. ed. -: Pearson, 2014.

UG, Sascha Nitsch Unternehmensberatung. Sockets Tutorial. Disponível em: <http://www.linuxhowtos.org/C_C++/socket.htm>. Acesso em: 08 nov. 2016.

APÊNDICE A - IMPLEMENTAÇÃO DO SERVIDOR USANDO ZeroMQ PARA MEDIR O TEMPO DE RESPOSTA

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "../include/zmq.h"
#include <assert.h>

int main(int argc, char *argv[]) {
    struct timeval inicio, final;
    long int tmicro;

    zmq_msg_t msg;
    int rc;
    void *ctx = zmq_ctx_new();
    void *responder = zmq_socket(ctx, ZMQ_REP);
    zmq_bind(responder, "tcp://*:5551");
    rc = zmq_msg_init(&msg);
    int count = 0;

    while (count < 3600 * atof(argv[2])) {
        gettimeofday(&inicio, NULL);
        zmq_recvmsg(responder, &msg, 0);
        zmq_sendmsg(responder, &msg, 0);
        gettimeofday(&final, NULL);
```

```
            tmicro = (long int) (1000000 * (final.tv_sec - inicio.tv_sec) + (final.tv_usec - inicio.tv_usec));

        printf("%ld\n", tmicro);

        count += 1;

    }

    zmq_msg_close(&msg);

    zmq_close(responder);

    return 0;

}
```

APÊNDICE B - IMPLEMENTAÇÃO DO SERVIDOR USANDO BERKELEY PARA MEDIR O TEMPO DE RESPOSTA

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h>

void error(const char *msg) {
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[]) {
    struct timeval inicio, final;
    long int tmicro;

    int sockfd, newsockfd, portno;
    socklen_t clilen;
    char buffer[120000];
    struct sockaddr_in serv_addr, cli_addr;
    int n;
```

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
   error("ERROR opening socket");
bzero((char *) &serv_addr, sizeof(serv_addr));
portno = atoi(argv[1]);
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);

if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)
   error("ERROR on binding");
listen(sockfd, 5);
clilen = sizeof(cli_addr);
newsockfd = accept(sockfd,
                (struct sockaddr *) &cli_addr,
                &clilen);
if (newsockfd < 0)
   error("ERROR on accept");
bzero(buffer, 120000);
int count = 0;
while (count < 3600 * atof(argv[2])) {
   gettimeofday(&inicio, NULL);
   n = read(newsockfd, buffer, 99999);
   if (n < 0) error("ERROR reading from socket");
   n = write(newsockfd, "I got your message", 18);
```

```
        if (n < 0) error("ERROR writing to socket");
        gettimeofday(&final, NULL);
            tmicro = (long int) (1000000 * (final.tv_sec - inicio.tv_sec) + (final.tv_usec - inicio.tv_usec));
        printf("%ld\n", tmicro);
        count += 1;
    }
    close(newsockfd);
    close(sockfd);
    return 0;
}
```

APÊNDICE C - IMPLEMENTAÇÃO DO SERVIDOR USANDO NANOMSG PARA MEDIR O TEMPO DE RESPOSTA

```
#include "../src/nn.h"
#include "../src/tcp.h"
#include "../src/pair.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../src/utils/err.c"
#include "../src/utils/sleep.c"
#include <sys/time.h>

int main(int argc, char *argv[]) {
    struct timeval inicio, final;
    const char *bind_to;
    size_t sz;
    int rts;
    char *buf;
    int nbytes;
    int s;
    int rc;
    int i;
    int opt;

    bind_to = argv[1];
    sz = atoi(argv[2]);
```

```
rts = atoi(argv[3]);

s = nn_socket(AF_SP, NN_PAIR);
nn_assert(s != -1);
opt = 1;
rc = nn_setsockopt(s, NN_TCP, NN_TCP_NODELAY, &opt, sizeof(opt));
nn_assert(rc == 0);
opt = -1;
rc = nn_setsockopt(s, NN_SOL_SOCKET, NN_RCVMAXSIZE, &opt, sizeof(opt));
nn_assert(rc == 0);
opt = 1000;
rc = nn_setsockopt(s, NN_SOL_SOCKET, NN_LINGER, &opt, sizeof(opt));
nn_assert(rc == 0);
rc = nn_bind(s, bind_to);
nn_assert(rc >= 0);

buf = malloc(sz);
nn_assert(buf);
memset(buf, 111, sz);

long int tmicro = 0;

for (i = 0; i != rts; i++) {
    gettimeofday(&inicio, NULL);
    nbytes = nn_recv(s, buf, sz, 0);
    nn_assert(nbytes == (int) sz);
```

```
        nbytes = nn_send(s, buf, sz, 0);
        nn_assert(nbytes == (int) sz);
        gettimeofday(&final, NULL);
            tmicro = (long int) (1000000 * (final.tv_sec - inicio.tv_sec) + (final.tv_usec - inicio.tv_usec));
        printf("%ld\n", tmicro);
    }
    free(buf);
    rc = nn_close(s);
    nn_assert(rc == 0);
    return 0;
}
```