CENTRO PAULA SOUZA

Faculdade de Tecnologia de Americana
Curso Superior de Bacharelado em Análise de Sistemas e Tecnologia da Informação

DESENVOLVIMENTO PARA ARQUITETURA DISTRIBUÍDA UTILIZANDO REMOTE PROCEDURE CALL (RPC)

BRUNO EDUARDO VON AH

Americana, SP
2013

Faculdade de Tecnologia de Americana
Curso Superior de Bacharelado em Análise de Sistemas e Tecnologia da Informação

DESENVOLVIMENTO PARA ARQUITETURA DISTRIBUÍDA UTILIZANDO REMOTE PROCEDURE CALL (RPC)

BRUNO EDUARDO VON AH

brunoevonah@hotmail.com

Trabalho de Conclusão de Curso, apresentado à Faculdade de Tecnologia de Americana como parte das exigências do curso de Análise de Sistemas e Tecnologia da Informação para a obtenção do título de Bacharel em Sistemas e Tecnologia da Informação.

Orientador: Prof. Dr. José Luís Zem

Área: Computação distribuída, arquitetura de computadores, RPC.

Americana, SP
2013

BRUNO EDUARDO VON AH RA: 0922806

DESENVOLVIMENTO PARA ARQUITETURA DISTRIBUÍDA UTILIZANDO REMOTE PROCEDURE CALL (RPC)

Trabalho de Conclusão de Curso aprovado como requisito parcial para obtenção do título de Bacharel em Análise de Sistemas no curso de Análise de Sistemas e Tecnologia da Informação da Faculdade de Tecnologia de Americana.

Banca Examinadora

Orientador: José Luís Zem, Doutor, Faculdade de Tecnologia de Americana.

Assinatura:____

Convidado: Eduardo Antônio Vicentini, Mestre, Faculdade de Tecnologia de Americana.

Assinatura:____

Convidado: Rogério Nunes de Freitas, Especialista, Faculdade de Tecnologia de Americana.

Assinatura:____

AMERICANA/SP 17/06/2013

Dedico este trabalho a todos aqueles que acreditaram e ainda acreditam em mim, em especial minha mãe Elisabete, meu pai Antônio Carlos e meu irmão Arthur.

AGRADECIMENTOS

Aos meus amigos e companheiros de classe Bruno Campos de Azevedo, Diego Henrique Dias e Joel Vasconcelos Junior, que estiveram comigo e me apoiaram durante toda esta trajetória.

Aos demais colegas de classe que proporcionaram um ambiente descontraído e também me auxiliaram durante esses quatro anos de desenvolvimento pessoal, profissional e acadêmico.

Ao Prof. Dr. José Luís Zem quem me orientou para construir este trabalho, além de me apresentar uma nova visão da área de Tecnologia da Informação e teve muita calma no decorrer de todo o processo de criação deste trabalho.

Agradeço a todos os demais envolvidos direta e indiretamente, que me deram forças para seguir em frente e chegar aqui.

"*A ciência nunca resolve um problema sem criar pelo menos outros dez.*" *(George Bernard Shaw)*

RESUMO

O presente texto conceitua o fato do grande aumento de informações geradas diariamente e a questão das limitações dos computadores convencionais em processá-las, direcionando assim esforços para os sistemas distribuídos, uma das soluções empregadas para lidar com este fluxo de dados crescente, o que aumentou a importância da computação distribuída. Como objetivo geral estipulou-se o estudo de tais sistemas, em especial a arquitetura dos computadores que os formam, assim como do desenvolvimento de aplicações capazes de lidar com essas estruturas, o qual neste trabalho é através das técnicas de RPC. Para aprofundamento do tema foi realizado a criação de uma aplicação que faz uso de tal abordagem, assim como um estudo comparativo entre o desempenho do modelo convencional de execução, o sequencial local, e o distribuído sequencial. A metodologia empregada foi da pesquisa bibliográfica para obtenção das informações que foram expostas e analisadas, aliada a pesquisa experimental em um ambiente virtual para a execução dos algoritmos desenvolvidos. Durante o experimento foram testadas três diferentes situações de execução de um algoritmo *quick sort* sendo uma delas a convencional, e nas outras duas distribuídas, porém uma na mesma máquina e outra em máquinas separadas. Foi então observado dados que demonstraram um desempenho similar entre uma aplicação distribuída com RPC e uma com execução sequencial. Baseado no estudo da literatura e na observação da prática foi concluído que o ambiente virtual provavelmente favoreceu a aplicação RPC, o que fez com que fosse proposto outros estudos em outras situações para a determinação dos impactos no desempenho. No entanto mesmo que haja degradação no comportamento, foi observado que as técnicas RPC são muito poderosas e valiosas para, as cada vez mais heterogêneas, plataformas computacionais como os *clusters*, grades e nuvens, representantes de sistemas distribuídos de alto nível.

Palavras Chave: Computação distribuída, RPC, arquitetura de computadores.

ABSTRACT

The present text conceptualizes the fact of a major growth in the daily information creation and the issue of conventional computers limitation to process it, in this way directing efforts to the distributed systems, one of the solutions employed to deal with that increasing data flow, which has risen the importance of distributed computing. As a general objective it was stipulated the study of said systems, specially of its computers architecture that shapes them, as well of the development of applications capable to deal with these structures, which in this work is through the RPC techniques. To develop further the theme was created an application that can make use of such approach, just as a comparative study between the conventional execution model performance, the local sequential, and the sequential distributed. The methodology used was literature research to obtain information which were exposed and analyzed with the experimental research in a virtual environment to execute the developed algorithms. During the experiment it was used three different situations for the quick sort algorithm execution, being one of them the conventional, and in the other two the distributed, however one in the same machine and the other in separated machines. Then it was observed data which shows a similar performance between a distributed software with RPC, and one with sequential execution. Based on the literature research and the observation of the practice it was concluded that the virtual environment probably helped the RPC application, which resulted in the proposition of others studies in others situations to determine the performance impacts. Although even that exist a behavior degradation, it was observed that RPC techniques are very powerful and valuable to the, increasingly heterogeneous, computational platforms as clusters, grids and clouds, the high level's distributed systems representatives.

Keywords: Distributed computing, RPC, computers architecture

LISTA DE FIGURAS

Figura 01: Nível de acoplamento..17
Figura 02: Cluster genérico...24
Figura 03: Grade computacional.. 25
Figura 04: Computação na Nuvem..31
Figura 05: Comunicação RPC convencional.. 36
Figura 06: Execução de uma chamada de procedimento remota.........................37
Figura 07: Processo de chamada RPC convencional em relação ao tempo........ 39
Figura 08: Comparação entre RPC tradicional e RPC assíncrona........................40
Figura 09: Modelo de protocolo RPC ONC RPC..43
Figura 10: Intervalos de numeração do programa ONC RPC...............................43
Figura 11: Exemplo de protocolo RPC ONC RPC...44
Figura 12: Exemplo rotina simplificada *rpc_call*.. 46
Figura 13: Exemplo rotina de alto nível *clnt_*create...47
Figura 14: Protocolo Calculadora RPC.. 48
Figura 15: Dados importantes biblioteca Calculadora RPC.................................. 49
Figura 16: Exemplo para aplicação Calculadora RPC Cliente e Servidor.............50
Figura 17: Compilando aplicação Calculadora RPC Cliente e Servidor................52
Figura 18: Executando as aplicações Calculadora RPC.......................................53

LISTA DE SIGLAS

ACK *Acknowledge* - Reconhecimento.

ADSC Análise e Desempenho de Sistemas Computacionais

API *Application Programming Interface* - Interface para programação de aplicação.

COMA *Cache Only Memory Access* - Acesso somente a memória cache.

CPU *Central Processing Unit* - Unidade central de processamento.

daemon *Disk And Execution Monitor* - Monitor de execução e disco.

DCE *Distributed Computing Environment* - Ambiente de computação distribuída.

DNA/ADN *Deoxyribonucleic Acid - Ácido Desoxirribonucléico*

Gbps *Gigabits* por segundo

GCC *GNU Compiler Collection* - Coleção de compilador GNU

GLIBC *GNU C Library* - Biblioteca C GNU

GNU *GNU's Not Unix* - GNU não é Unix.

HPC *High Performance Computing* - Computação de alto desempenho.

IaaS *Infrastructure as a Service* - Infraestrutura como serviço

IDL *Interface Definition Language* - Linguagem de definição de interface.

IP *Internet Protocol* - Protocolo Internet.

MB *Megabytes.*

MIDL *Microsoft Interface Definition Language* - Linguagem de definição de interface Microsoft.

MPI *Message Passing Interface* - Interface para passagem de mensagens.

MPP *Massively Parallel Processors* - Processadores massivamente paralelos.

NCS *Network Computing System* - Sistema computacional em rede.

NDR *Network Data Representation* - Representação de dados de rede.

NFS *Network File System* - Sistema de arquivos de rede.

NIS *Network Information Service* - Serviço de informação da rede.

NoSQL *Not only SQL* - Não apenas SQL.

NUMA *NonUniform Memory Access* - Acesso não uniforme a memória.

ONC RPC *Open Network Computing Remote Procedure Call* - Chamada de procedimento remoto da rede aberta de computação.

OpenCL *Open Computing Language* - Linguagem de computação aberta.

OV Organização Virtual.

PaaS *Platform as a Service* - Plataforma como serviço.

RPC *Remote Procedure Call(ing)* - Chamada de procedimento remoto.

SaaS *Software as a Service* - Software como serviço.

SLA *Service Level Agreement* - Acordo de nível de serviço

SO Sistema Operacional

TCP *Transmission Control Protocol* - Protocolo de controle de transmissão

UDP *User Datagram Protocol* - Protocolo de datagramas de usuário

UMA *Uniform Memory Access* - Acesso uniforme à memória

VLIW *Very Long Instruction Word* - Palavra de instrução muito longa.

XDR *eXternal Data Representation* - Representação de dados externa.

SUMÁRIO

1 INTRODUÇÃO 14

2 ARQUITETURA DE COMPUTADORES DISTRIBUÍDOS E PARALELOS 17

2.1 PARALELISMO NO CHIP 18

2.2 CO-PROCESSADORES 18

2.3 MÚLTI PROCESSADORES DE MEMÓRIA COMPARTILHADA 19

2.4 MÚLTI COMPUTADOR 20

2.5 GRADE COMPUTACIONAL 21

2.6 PRINCIPAIS PLATAFORMAS COMPUTACIONAIS 22

2.6.1 CLUSTERS 22

2.6.2 GRADE COMPUTACIONAL (*GRID*) 24

2.6.3 COMPUTAÇÃO NA NUVEM 27

3 REMOTE PROCEDURE CALL - RPC 32

3.1 FUNCIONAMENTO E ELEMENTOS DE UM SISTEMA RPC 33

3.2 DESENVOLVIMENTO COM RPC 41

3.2.1 DEFINIÇÃO DO PROTOCOLO 42

3.2.2 INTERFACE RPC 45

3.2.3 ELABORAÇÃO DAS APLICAÇÕES CLIENTE E SERVIDOR. 48

4 ESTUDO DE CASO 54

4.1 PRIMEIRO CASO - EXECUÇÂO SEQUENCIAL 56

4.2 SEGUNDO CASO - EXECUÇÃO RPC MESMA MÁQUINA 57

4.3 TERCEIRO CASO - EXECUÇÃO RPC MÁQUINAS DISTINTAS 57

5 DISCUSSÃO DOS RESULTADOS 59

6 CONSIDERAÇÕES FINAIS 60

7 REFERÊNCIAS 62

APÊNDICE A - Algoritmos do estudo de caso. 64

APÊNDICE B - Amostras do estudo de caso. 76

1 INTRODUÇÃO

Com a enorme quantidade de informação gerada e armazenada diariamente, devido em grande parte ao advento da internet, novas abordagens para sistemas computacionais têm sido estudadas com o intuito de prover meios capazes de atender a demanda crescente por poder computacional, além da obtenção de alta disponibilidade dos serviços providos por tais estruturas.

Entre os principais interessados nesses recursos estão os provedores de serviços da internet como o Google, Facebook, Amazon, entre outros que buscam atender grandes quantidades de usuários conectados ao mesmo tempo com um nível de qualidade aceitável, e têm-se também os grandes centros de pesquisas, sejam eles farmacêuticos, biológicos, astronômicos, físico-químicos, meteorológicos, etc. Que necessitam processar uma grande quantidade de informações que com computadores comuns poderiam levar anos ou até mesmo milênios (TANENBAUM; 2007, p.322).

Mesmo com máquinas cada vez mais rápidas, a demanda ainda é grande e tem crescido junto com a capacidade delas. Além disso, tem-se o fato de:

> "Embora as velocidades de relógio continuem subindo, a velocidade dos circuitos não podem aumentar indefinidamente. A velocidade da luz já é um grande problema para projetistas de computadores de alta tecnologia, e a perspectiva de conseguir que elétrons e fótons se movam com maior rapidez são desanimadoras. Questões de dissipação de calor estão transformando supercomputadores em condicionadores de ar de última geração. Por fim, como o tamanho dos transistores continua a diminuir, chegará um ponto em que cada transistor terá um número tão pequeno de átomos dentro dele que efeitos da mecânica quântica - por exemplo, o princípio da incerteza de Heisenberg - podem se tornar um grande problema (TANENBAUM; 2007, p.322)."

Tendo estes aspectos em mente os arquitetos de computadores têm dirigido suas atenções a uma solução que visa driblar estes problemas, oferecendo desempenho de uma forma diferente: os sistemas computacionais paralelos, uma especialização de sistemas distribuídos.

Um sistema paralelo é aquele em que pelo menos dois processadores e/ou elementos de processamento são capazes de trabalharem juntos em uma mesma tarefa, cada um fazendo uma parte ou replicando-a, seja para aumentar o desempenho, afirmar confiabilidade ao resultado obtido ou garantir disponibilidade contínua para a ação que esta sendo realizada. No entanto um sistema distribuído é aquele onde pode-se distribuir tarefas e atividades entre diversos recursos computacionais independentes, não sendo estas executadas necessariamente em paralelo, portanto todo sistema paralelo é distribuído, porém não necessariamente todo distribuído é paralelo (TANENBAUM; 2007, p.322).

Uma possível confusão pode ocorrer entre um sistema distribuído e uma rede de computadores como explica Tanenbaum (2003, p.2):

> "A principal diferença entre eles[sistemas distribuídos e redes]é que, em um sistema distribuído, um conjunto de computadores independentes parece ser, para usuários, um único sistema coerente. Em geral, ele tem um único modelo ou paradigma que apresenta aos usuários. Com freqüência, uma camada de software sobre o sistema operacional, chamada *middleware*, é responsável pela implementação desse modelo. Um exemplo bem conhecido de sistema distribuído é a *World Wide Web*, na qual tudo tem a aparência de um documento (uma página da Web)."

O trabalho se justificou pois é com esta arquitetura que o enorme fluxo de dados diários que passam por grandes provedores de serviços na internet, ou então grandes projetos científicos com um grau de complexidade elevado, têm sido capazes de existirem e principalmente processar toda essa informação. Os sistemas distribuídos são os responsáveis por grande parte do avanço da tecnologia atual e de sua expansão, afinal seria inviável disponibilizar o acesso aos mais variados serviços para a quantidade cada vez maior de usuários, assim como o mapeamento do DNA/ADN com toda sua complexidade jamais chegaria ao estado em que hoje se encontra sem o poder computacional disponibilizado por estes sistemas.

A pesquisa quanto ao seu objetivo é exploratória que segundo Silva e Menezes (2001, p. 21) *"[...]visa proporcionar maior familiaridade com o problema com vistas a torná-lo explícito ou a construir hipóteses. Envolve levantamento bibliográfico.",* sendo neste caso os sistemas distribuídos e o desenvolvimento de aplicações com RPC para estas plataformas o problema a ser familiarizado.

Com a familiarização do tema como objetivo geral, os objetivos específicos ficam em como desenvolver uma aplicação distribuída utilizando-se da técnica de RPC e então realizar uma comparação com um software de mesmo propósito porém convencional, ou seja, sequencial com execução local para determinar diferenças de desempenho.

Para reunir informações será utilizado o método da pesquisa bibliográfica, pois será *"[...] elaborada a partir de material já publicado, constituído principalmente de livros, artigos de periódicos e atualmente com material disponibilizado na Internet"* (Id. Ibid.) a fim de obter as informações necessárias sobre a arquitetura dos sistemas distribuídos e a utilização de RPC para acessar tais sistemas. Combinado será utilizado a pesquisa experimental que "*[...] se determina um objeto de estudo, selecionam-se as variáveis que seriam capazes de influenciá-lo [...]"* (Id. Ibid.), em um ambiente virtual criado para experimentações e simulações através de máquinas virtuais.

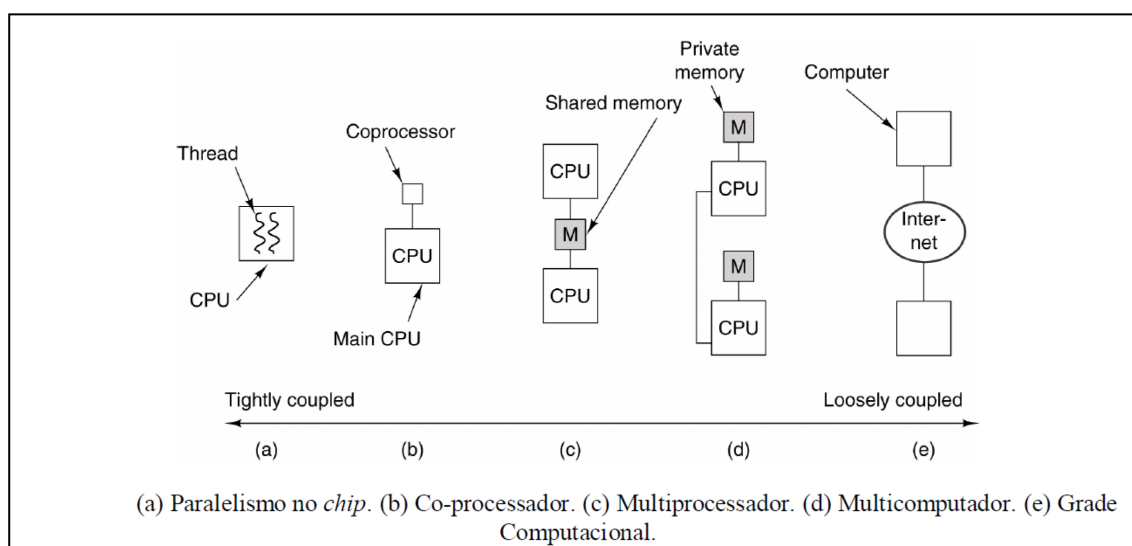
A análise das informações será feita através da metodologia indutiva com *"[...]a generalização deriva de observações de casos da realidade concreta" (*Id. Ibid., p. 26), utilizando portanto como base as informações adquiridas e observadas através da experimentação e embasamento teórico para chegar a uma conclusão.

Por fim a apresentação das conclusões serão feitas de forma dissertativa, através de tabelas comparativas e de apresentação prática.

2 ARQUITETURA DE COMPUTADORES DISTRIBUÍDOS E PARALELOS

Quanto a forma de implementar a arquitetura paralela ou distribuída existem diversas, e de acordo com a escolha ela terá uma característica especial de grande importância chamada acoplamento. Quando os componentes estão próximos uns dos outros (em termos computacionais), havendo alta largura de banda e rápido tempo de resposta, tem-se os sistemas fortemente acoplados. Já a contrapartida que é definida por baixa largura de banda, alto tempo de resposta e localização remota uns dos outros, caracterizam-se então os fracamente acoplados (TANENBAUM; 2007, p.323).

Na Figura 01 é possível ver as diferentes implementações desta arquitetura, junto a uma comparação do nível de acoplamento. Nesta figura da esquerda para a direita ocorre o enfraquecimento, portanto, sendo o extremo esquerdo as fortemente acopladas e no oposto as fracamente.



(a) Paralelismo no *chip*. (b) Co-processador. (c) Multiprocessador. (d) Multicomputador. (e) Grade Computacional.

Figura 01 - Nível de acoplamento **Fonte:Tanenbaum (2007, p.323)**

2.1 PARALELISMO NO CHIP

O paralelismo encontrado no nível mais baixo é o realizado no chip, que tem o intuito de fazê-lo realizar mais de uma ação ao mesmo tempo, por consequência elevando a velocidade de execução de tarefas (TANENBAUM, 2007).

Entre as técnicas que costumam ser implementadas estão (TANENBAUM, 2007):

- Paralelismo no nível de instrução: Atua através da emissão de várias instruções por ciclo do *clock* (ou relógio), podem ser encontradas em processadores VLIW (*Very Long Instruction Word - Palavra de instrução muito longa*) e nos superescalares;

- *Multithreading*: Funciona fazendo com que a CPU possa ter mais de uma *thread* de controle concorrente, ocasionando com que ao ocorrer a parada de uma delas devido a uma requisição que demande tempo a outra entre em funcionamento, com isso deixando-a ocupada o máximo possível. Existem nas variações chamadas de granulação fina, grossa e *multhreading* simultâneo.

- Múltiplas CPU*s* no mesmo chip: Com a adição de uma ou mais CPU*s* (homogêneas ou heterogêneas) no mesmo chip é obtido essa forma de paralelismo. Possui desempenho e custo maior que o *multithreading*, e costuma ser utilizado em conjunto com ela em processadores de computadores tradicionais topo de linha.

2.2 CO-PROCESSADORES

Reduzindo um pouco o acoplamento, porém ainda mantendo em um nível alto, estão os co-processadores. Nessa abordagem é inserido ao sistema uma ou mais CPU*s*, porém todas estas são especializadas em uma determinada função que costuma ser necessária ao funcionamento do conjunto, diferentemente da principal que é de propósito geral (TANENBAUM, 2007).

Com a utilização desses itens, é possível aumentar o desempenho agregando o paralelismo uma vez que o processamento especializado é executado mais rápido do que ao ser realizado por uma fonte de cunho geral.

Entre os diversos co-processadores existentes um dos mais utilizados é o de rede para computadores de uso geral, mas também há os de imagem, som, vídeo, ponto flutuante, criptografia entre outros que estão mais presentes em eletroeletrônicos que têm uso específico.

2.3 MÚLTI PROCESSADORES DE MEMÓRIA COMPARTILHADA

Estando no meio termo no quesito acoplamento estão os sistemas com múlti processadores de memória compartilhada. Neste método são adicionadas uma ou mais CPUs de propósito geral (normalmente homogêneas) de forma parecida com os co-processadores, exceto pelo fato de nessa serem utilizadas unidades totalmente desenvolvidas e não ser possível dizer qual é a principal, uma vez que qualquer uma pode ser (TANENBAUM, 2007).

Entre os múlti processadores pode-se implementar a memória compartilhada de três formas (TANENBAUM, 2007):

- *UMA (Uniform Memory Access - Acesso uniforme à memória)*: Sua característica é o fato de o tempo de acesso aos módulos de memória ser igual para todas as CPUs, fazendo com que o desempenho seja controlado o que auxilia no desenvolvimento de código paralelo.

- *NUMA (NonUniform Memory Access - Acesso não uniforme à memória)*: Diferentemente da UMA cada CPU possui um tempo de acesso diferente a cada módulo de memória, o que faz com que o local dos dados e aonde será executada a tarefa tenha uma grande influência no desempenho.

- *COMA (Cache Only Memory Access - Acesso somente a memória cache)*: Assim como as máquinas NUMA não há acesso uniforme a memória, e neste caso a

memória de todo o sistema é utilizado como uma grande cache, fazendo com que as linhas de cache fiquem migrando pelo sistema conforme a necessidade.

2.4 MÚLTI COMPUTADOR

Alterando o modelo de programação e um pouco a forma de implementar o paralelismo tem-se os sistemas múlti computador, onde vários computadores (não apenas CPUs como nos multiprocessadores) são interligados com o intuito de aumentar o desempenho em tarefas. Normalmente os múlti processados fazem parte de um múlti computador, onde cada um deles é chamado de nó (TANENBAUM, 2007).

Estes nós podem, por sua vez, serem interconectados utilizando redes de alta velocidades, muitas vezes proprietárias para aquele sistema específico, ou até mesmo por redes comum do tipo *Ethernet*. Apesar da velocidade que pode ser atingida por estas redes, esses conjuntos acabam tendo um acoplamento fraco, pois em comparação com o acesso a memória ou ainda ao barramento da cache, sua resposta não é tão ágil.

Os múlti computadores são encontrados em duas categorias:

- *MPPs (Massively Parallel Processors - processadores maciçamente paralelos)*: Podem também serem denominados supercomputadores, utilizados em centros de pesquisas que envolvem cálculos de alto nível de complexidade e quantidades exorbitantes deles, são encontrados nos dias de hoje no ambiente corporativo/comercial como em bancos e outros segmentos que demandam grande quantidade de dados a serem processados (TANENBAUM; 2007, p.359).

Característica desse tipo de sistema é o fato de utilizarem redes de altas velocidades proprietárias, assim como várias bibliotecas e softwares, que aliados a um robusto sistema de E/S (entrada e saída) são capazes de mover quantidades significativas de dados em uma velocidade expressiva, o que se traduz em um alto valor investido para a construção deste (TANENBAUM; 2007, p.359).

- *Clusters*: Funcionam e se estruturam basicamente como os MPPs, porém a diferença fundamental é que estes são construídos utilizando computadores domésticos ou estações de trabalho e conectados por uma rede comum do mercado. Como reflexo desse modelo, possuem o desempenho inferior aos MPPs e como consequência o valor a ser investido também, porém nos últimos anos redes de alta velocidade têm sido mais acessíveis e não necessariamente específicas a um sistema, elevando assim os *clusters* e os tornando mais atraentes (TANENBAUM; 2007, p.359).

Os *clusters* têm ainda como característica o fato de não necessariamente serem padronizados como são os MPPs, assim podem conter diversos computadores heterogêneos em sua formação.

2.5 GRADE COMPUTACIONAL

Atingindo o acoplamento de nível mais fraco estão as grades computacionais, estas consistem, diferentemente dos *clusters*, em computadores espalhados não somente no mesmo edifício, mas sim por praticamente qualquer lugar do mundo, normalmente de várias corporações, institutos e faculdades, que juntos possuem uma mesma meta e que têm intenção de cooperarem para atingi-la (TANENBAUM; 2007, p.380).

Não só apenas compartilhando a infra-estrutura computacional, a grade provém também outros recursos, como conhecimento e experiências que os participantes queiram dividir.

Outra característica importante são os diferentes níveis de acesso e disponibilidade encontrados, cada organização define o que e como irá compartilhar e por quanto tempo, e as vezes caso ela precise daquele recurso, a prioridade será direcionada para o uso interno (TANENBAUM; 2007, p.380).

2.6 PRINCIPAIS PLATAFORMAS COMPUTACIONAIS

Dentre as plataformas computacionais existentes (computadores, unidades de processamento, celulares, entre outros), para este trabalho as que têm uma maior importância são: *Clusters* de computadores, Grades computacionais e a Computação na Nuvem.

Todas estas são representações de sistemas distribuídos que segundo Tanenbaum e Steen (2007, p.1), podem ser definidos como: "[...] um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente." Cada um desses modelos possuem suas características próprias e as similares aos outros, fazendo-se de grande importância o estudo simultâneo deles, para elucidar o funcionamento e a coexistência que apresentam.

2.6.1 CLUSTERS

Os *clusters* foram criados com o objetivo de atender a demanda por computação de alto desempenho a um baixo custo, em especial para resolver problemas das áreas da ciência, engenharia e até mesmo negócios privados que não podiam ser efetivamente resolvidos com a utilização de supercomputadores, já que seu custo iria inviabilizar grande parte dos projetos (STERLING, 2001).

Pode-se entender *cluster* com base em Sadashiv e Kumar (2011) como sendo uma coleção de computadores paralelos ou distribuídos que estão interconectados através de redes de alta velocidade tais como *Myrinet*, *Infiniband* e até mesmo *Ethernet*. Este conjunto trabalha unido na execução de tarefas com processamento e dados intensivos, que seriam inviáveis para um único computador convencional. Um modelo genérico de cluster pode ser observado na Figura 02.

Possuem como principais propósitos a alta disponibilidade, balanceamento de carga e processamento de alto desempenho (processamento paralelo), podem também serem utilizados para alta confiabilidade.

Através da replicação de nós é obtido a alta disponibilidade, uma vez que ao ocorrer a falha de um componente do sistema, outro assume o seu lugar. Assim há

também um ganho de desempenho ao eliminar o impedimento que seria causado pela falta de um nó em um único ponto.

Por possuir múltiplos computadores interconectados em um *cluster*, eles podem dividir e compartilhar as tarefas como se fosse um único sistema, uma única máquina virtual, tornando possível que haja o balanceamento da carga computacional a ser executada entre os nós componentes do sistema e elevando a velocidade para executar funções.

A alta confiabilidade é atingida ao fazer-se com que uma mesma tarefa seja executada paralelamente em diferentes nós, acarretando que o resultado final seja calculado por mais de uma fonte, logo sendo possível confirmar a segurança da conclusão.

Alguns dos desafios encontrados na computação em *cluster* estão relacionados segundo Sadashiv e Kumar (2011) com:

- *Middleware*: Produzir ambientes de software capaz de prover a ilusão que se trata de apenas um único sistema, ao invés de uma coleção de computadores independentes.

- Programação: As aplicações devem ser explicitamente escritas para determinado *cluster*, devendo incorporar a divisão de tarefas entre os nós computacionais além da comunicação que será utilizada, tornando-as muito específicas.

- Elasticidade: Controle sobre a variação em tempo real do tempo de resposta quando houver dramático aumento do número de requisições de um ou mais serviços, garantindo a qualidade do mesmo.

- Escalabilidade: Ser capaz de adequar-se aos requerimentos adicionais de um determinado recurso e assim afetando o desempenho do sistema.

Algumas das aplicações para os *clusters*, de acordo com Sadashiv e Kumar (2011), são resolver aplicações de alta complexidade tais como modelagem do clima, simulações de colisões de automóveis, ciências da vida, processamento dinâmica de fluídos, simulações nucleares, processamento de imagens, eletromagnetismo, mineração de dados, aerodinâmica e astrofísica. Eles também são utilizados em aplicações comercias como no setor bancário para alcançar alta disponibilidade e *backup*. *Clusters* são utilizados para hospedar vários novos serviços da internet como Hotmail, aplicações *web*, banco de dados e outras aplicações comerciais.

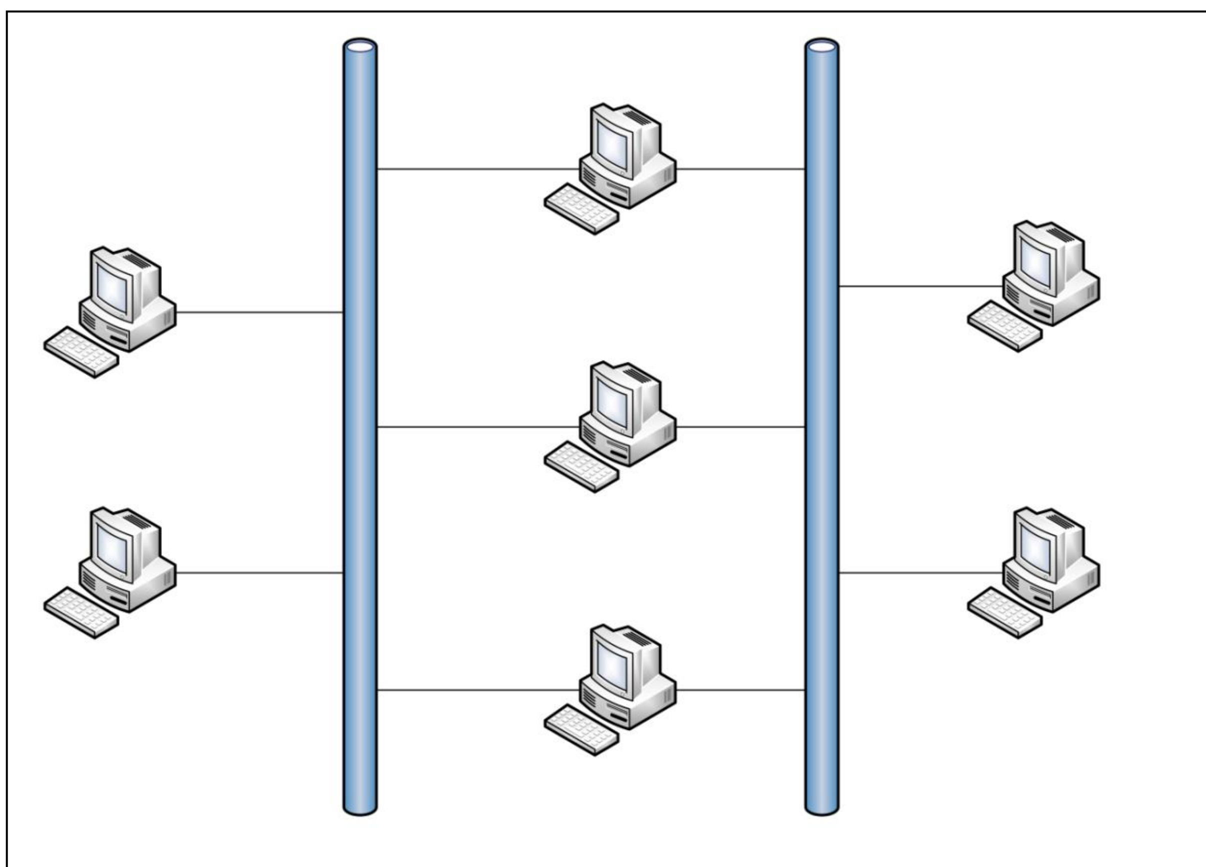


Figura 02 - Cluster genérico. **Fonte: Criada pelo autor.**

2.6.2 GRADE COMPUTACIONAL (*GRID*)

Diferente da forma convencional de computação distribuída, as grades (*Grid* em inglês) com o seu foco em larga escala de compartilhamento de recursos, aplicações inovadoras e orientação ao alto desempenho, teve a utilização desse termo pela primeira vez em meados de 1990 para denotar uma infra-estrutura para ciência e engenharia avançadas (FOSTER; KESSELMAN; TUECKE, 2001).

Segundo Buya et. al. (2002) citado por Sadashiv e Kumar (2011, p.478) *Grid* é definido como um tipo de sistema paralelo e distribuído que viabiliza a partilha, seleção e agregação de recursos autônomos geograficamente distribuídos em tempo de execução dependendo de sua disponibilidade, capacidade, desempenho, custo e requerimentos de qualidade de serviço por usuários.

As Grades têm como objetivo, de acordo com Foster, Kesselman e Tuecke (2001), prover de forma flexível, segura e coordenada a divisão de recursos e resolução de problemas, em dinâmicas e múlti institucionais Organizações Virtuais, conforme pode ser observado na Figura 03, três instituições diferentes dividem seus recursos entre si através da internet, formando uma Grade.

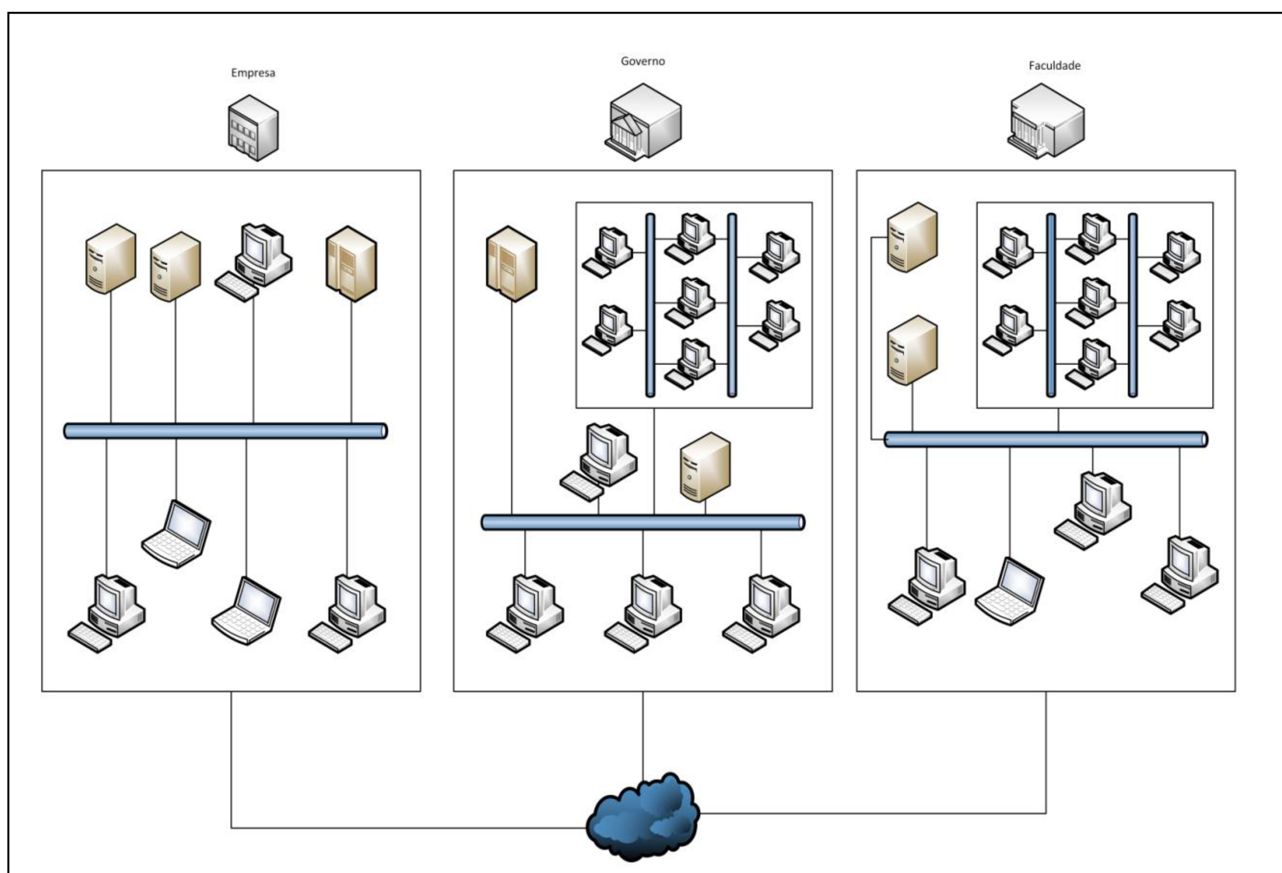


Figura 03 - Grade computacional. **Fonte: Criada pelo autor.**

Uma Organização Virtual (OV) é o nome dado a um conjunto de indivíduos ou instituições seguindo um rígido controle de acesso, em que é claramente determinado o que é, quando, para quem e quais as condições em que um recurso pode ser disponibilizado. As regras para cada divisão são dinâmicas podendo ser alteradas de acordo com a vontade da instituição que a detém realmente, garantido

a cada uma delas o controle total sobre seus recursos que estarão ou não a disposição da OV (FOSTER; KESSELMAN; TUECKE, 2001).

Um importante aspecto das Grades é que elas devem ser capazes de estabelecer novas relações de compartilhamento com qualquer participante em potencial, através de interoperabilidade, uma vez que estão inseridas em um ambiente interconectado com computadores heterogêneos. Isso é alcançado pelo fato de ter uma arquitetura de protocolos, os quais são utilizados para delimitar as formas de negociação, gerenciamento, estabelecimento e uso eficaz das relações de compartilhamento. Aliado aos protocolos de padrões abertos, são utilizadas ferramentas como interfaces para programação de aplicativos e kits de desenvolvimento de softwares que facilitam a abstração para criação de uma Grade utilizável (FOSTER; KESSELMAN; TUECKE, 2001).

As aplicações das Grades estão sempre ligadas com o objetivo em comum de várias organizações, que formam a OV para superá-lo, e de acordo com Sadashiv e Kumar (2011, p.480) costumam ser empregadas em projetos de:

- Manufatura avançada;
- Simulação de reservas de petróleo;
- Processamento de dinâmicas de fluídos;
- Pesquisas de partículas físicas;
- Física de alta energia nuclear:
- Modelagem do clima;
- Bioinformática;
- Observação da análise da natureza de terrenos;
- Base de dados científicas;

- Serviços populares da *web* científica;

Entre as principais dificuldades encontradas na implementação e manutenção das Grades computacionais está a dinamicidade que esta possui por trabalhar com diversos componentes heterogêneos, e ao fato de os participantes poderem ser voláteis. Sendo assim, devido a sua natureza é preciso um grande esforço para administrar as políticas tanto locais como as globais que estão presentes neste tipo de sistema. Com a alta diversidade encontrada neste ambiente é elevado o grau de complexidade para a produção de aplicações que possam se beneficiar da estrutura (SADASHIV; KUMAR, 2011).

É importante lembrar que devido a variedade de organizações e sistemas envolvidos, é comum encontrar *Clusters* como parte de uma Grade, que podem estar completamente ou parcialmente disponíveis de acordo com a vontade do proprietário participante da OV.

2.6.3 COMPUTAÇÃO NA NUVEM

Computação na Nuvem é definido por Buya et. al. (2009) apud Sadashiv e Kumar (2011), assim como na Grade, sendo um tipo de sistema paralelo e distribuído, porém consistindo de uma coleção de computadores interconectados e virtuais onde são dinamicamente disponibilizados e apresentados como um ou mais recursos unificados de computação, com base em acordos de nível de serviço.

Este modelo possibilita o acesso em praticamente qualquer lugar, de forma conveniente e sob demanda a uma gama de recursos computacionais configuráveis e compartilhados, onde são rapidamente concedidos e liberados com o menor esforço de gerenciamento ou interação com o provedor do serviço.

A Citrix Systems Inc. (2012) acredita que é importante delimitar o que não é Nuvem, de forma a impedir que haja correlações equivocadas, conforme a seguir:

- Não é um lugar: A Nuvem pode estar em qualquer lugar, seja em um *datacenter* próprio ou de outra organização, não podendo ser determinado com

precisão. O mais importante é compreender que ela é uma nova forma radical de entregar, consumir e adotar serviços de TI com maior agilidade, eficiência e custo eficaz do que as abordagens tradicionais;

- Não é restrição: Ela deve trazer flexibilidade para escolher a melhor virtualização, redes de acesso, soluções de armazenamento e hardware para adequar as necessidades do cliente. Para este fim elas precisam ser abertas e com código fonte aberto, permitindo a inter-relação com as outras nuvens existentes principalmente de parceiros.

- Não é virtualização de servidores: Apesar de alguns especialista em TI acreditarem, a Nuvem não é a próxima geração de virtualização de servidores. Utilizando o Google por exemplo, tem-se que houve a implantação de uma arquitetura de Nuvem feita através de infra-estrutura física. Já no caso da Amazon, embora tenha como base a tecnologia Xen de virtualização, o responsável por toda a arquitetura é encontrado na nova camada de software desenvolvida pela empresa.

Sendo assim, Nuvem é uma nova forma de prover, gerenciar e orquestrar recursos de infra-estrutura de um ou mais *datacenters*.

- Não é uma ilha: Devido a existência da Nuvem pública e privada (discutido mais a frente) é possível deduzir a necessidade da escolha de um dos modelos somente. Entretanto, como já dito anteriormente, ela não é um lugar para alocar serviços de TI e então perder toda a inter-conectividade e acesso. Aplicando uma abordagem que contemple tanto a pública como a privada de acordo com a necessidade de cada carga de trabalho garante uma melhor estratégia do uso da Nuvem.

Quanto a arquitetura dos níveis de serviço oferecidos pela Nuvem atualmente estão divididos em três, com base em Sun Microsystems Inc (2012):

- *Software as a Service (SaaS)* : Software como um serviço esta na mais alta camada e representa a disponibilização de uma aplicação inteira como um serviço.

Essa concessão é realizada sob demanda, onde uma única instância do software é executada na infra-estrutura do provedor e serve diversas organizações clientes

Entre os principais representantes de SaaS tem-se a SalesForce.com, Google Apps, Office 365, entre outros.

- *Platform as a Service (PaaS)* : A camada intermediária contém a plataforma como um serviço, onde é realizado o encapsulamento da abstração de um ambiente de desenvolvimento e o empacotamento de serviços necessários a seu funcionamento. O modelo padrão, encontrado como parte do *Amazon Web Services* por exemplo*,* é uma imagem Xen (máquina virtual) contendo um agrupamento básico da Web (distribuição Linux, um *Webserver* e um ambiente de programação tais como Pearl ou Ruby).

PaaS pode ser oferecida para todas as fases do desenvolvimento e teste de software ou podem ser especializadas em apenas uma área como gerenciamento de conteúdo. Como exemplo tem-se o Google *AppEngine* que utiliza a infra-estrutura do Google.

- *Infrastructure as a Service (IaaS)* : O nível mais baixo da Nuvem é formado pela infra-estrutura como um serviço, onde ocorre a entrega de armazenamento e processamento padronizado através da rede. Servidores, sistemas de armazenamento, switches, roteadores e outros sistemas são agregados (via tecnologia de virtualização normalmente) para serem capazes de lidar com tipos específicos de cargas variando desde processamento em cargas (*batch*) até picos de grande uso.

A mais famosa IaaS atualmente é o *Amazon Web Services*, com os serviços EC2 e S3, oferecendo base de infra-estrutura computacional e armazenamento escalável na Nuvem respectivamente. Atualmente outras IaaS já são oferecidas mesmo pela AWS como o Amazon RDS e AmazonDynamoDB que provém banco de dados na nuvens, sendo o segundo no modelo *NoSQL* (não relacionais).

Conforme citado anteriormente a computação na Nuvem pode se diferenciar devido ao seu tipo de acesso aos recursos que foram alocados, com isso tem-se a seguinte estrutura de acordo com Sun Microsystems Inc (2012) :

- Nuvem pública: São providas por terceiros, e as diferentes cargas de trabalhos de diversos clientes podem estar sendo executadas em um mesmo recurso ou grupo de recursos (servidores, sistemas de armazenamento, entre outros). Os usuários finais não têm conhecimento do que pode estar sendo processado junto a sua requisição.

- Nuvem privada: Nesta opção o cliente tem posse da infra-estrutura que estiver sendo solicitada sob-demanda, tendo total controle do que é executado , em qual recurso e quem pode ter acesso. Companhias lidando com proteção de dados e SLAs(*Service Level Agreements*, Acordos de nível de serviço) preferem esta opção.

- Nuvem Híbrida: Devido a constante integração entre diversos sistemas, é de grande importância que seja possível executar aplicações em vários ambientes, visando isto a nuvem híbrida combina a privada e a pública através da utilização tanto de infra-estrutura própria como compartilhada, porém de forma controlada, assim como as Grades mantendo a capacidade de prover escalonamento externo e sob-demanda mas ao mesmo tempo determinando como distribuir as aplicações entre este diferentes ambientes.

Com suas diversas arquiteturas e modelos as Nuvems costumam ter como objetivo hospedar conteúdos web, multimídia, além de serviços de computação de alto desempenho (*HPC - High Perfomance Computing*), armazenamento distribuído, integração múlti empresarial, entre outros (SADASHIV; KUMAR, 2011)

Para que seja possível manter toda essa estrutura funcionando alguns desafios são listados por Sadashiv e Kumar (2011) como a escalabilidade dinâmica, onde é necessário ajustar a quantidade de nós de computação de acordo com o tempo de resposta dos usuários. Há também a padronização, uma vez que cada corporação costuma ter sua própria interface para desenvolvimento de aplicações (API) além de protocolos próprios, com isso dificultando a interoperabilidade e

integração, assim como acontece na Grade. Outra importante questão é a da energia, uma vez que grande quantidade dela é demandada para atender as necessidades dos usuários.

Por fim, devido as proporções que uma nuvem pode atingir, aliado aos diferentes serviços que prove, é possível e até comum encontrar em sua infra-estrutura as plataformas citadas anteriormente, *Clusters* e Grades, como pode ser observado na Figura 04 ilustrando uma possível forma desta estrutura.

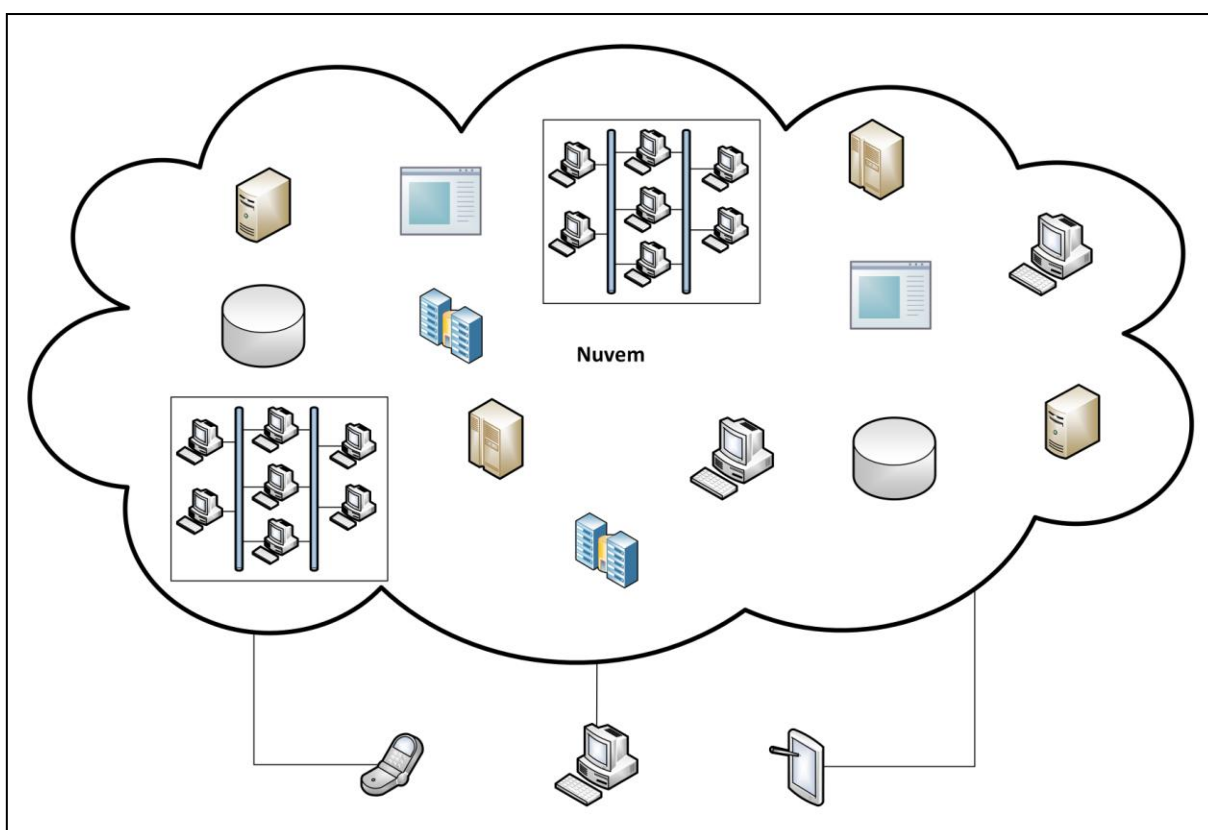


Figura 04 - Computação na Nuvem. **Fonte: Criada pelo autor.**

3 REMOTE PROCEDURE CALL - RPC

Para se ter acesso as estruturas e plataformas citadas anteriormente, com todo o seu poder e capacidade computacional, de forma organizada, eficiente e facilitada é preciso empregar técnicas e tecnologias de desenvolvimento distribuído. Afinal somente através destas é possível fazer com que um aplicativo seja executado fora de seu atual espaço de endereçamento, diferentemente de como ocorre na programação convencional, ou então seja facilitado o controle de execução em um espaço de memória compartilhado.

Existem diversas formas de se produzir uma aplicação distribuída, como por exemplo a MPI (*Message Passing Interface - Interface para Passagem de Mensagens*) e OpenCL (*Open Computing Language - Linguagem de Computação Aberta*) contemplando técnicas avançadas capazes de abrangerem diversos computadores e não somente seus processadores como também os processadores gráficos, no caso da última. Conjuntamente existem as funções das linguagens de desenvolvimento como a *Thread*[1] e o *Fork*[2] que trabalham a nível de paralelismo no chip e multiprocessador. Ademais com a utilização de protocolos de rede é possível acessar recursos remotos, como feito em *clusters* e grades, e executar através deles diversas funções, como é feito no *Sockets*, e técnicas de troca de mensagens por exemplo (SILBERSCHATZ; GALVIN; GAGNE, 2008).

Neste trabalho será analisado e desenvolvido a implementação através da técnica de RPC (*Remote Procedure Call - Chamada de Procedimento Remoto)*, a qual segundo Bloomer (1992, p. xxv) é a habilidade de chamar procedimentos fora do atual espaço de endereçamento de memória da aplicação. Em outras palavras, um programa local pode executar um procedimento em uma máquina remota, passar dados para ela e recuperar o resultado, permitindo a você escrever uma aplicação distribuída que pode fazer uso dos recursos computacionais de uma rede de trabalho.

---

[1] Fluxo de controle independente, criado por um processo.
[2] Replicação do processo atual, como filho, do qual o chamou.

Já Ribeiro (2005, p.292) elucida o princípio desta técnica de uma forma um pouco mais completa como sendo:

> "[...] estender a noção da chamada de procedimento local para a chamada remota. Em uma chamada de procedimento local, um processo faz uma requisição para um outro processo e envia argumentos definidos. O processo que recebeu a chamada irá executar alguma operação e devolver o resultado para o processo que o chamou. A chamada de procedimento remota é similar, mas os processos podem estar no mesmo sistema ou em diferentes sistemas conectados em uma rede."

3.1 FUNCIONAMENTO E ELEMENTOS DE UM SISTEMA RPC

O seu funcionamento esta baseado na troca de mensagens entre os processos, sendo que um sistema RPC, isto é, um sistema capaz de fazer uso dela, é aquele que possui a coleção de softwares necessários para suportar a programação de chamadas remotas de procedimento, além de serviços essenciais em execução formando assim um subconjunto lógico de um ambiente computacional distribuído (BLOOMER, 1992).

Um sistema RPC é basicamente composto por uma máquina cliente com a aplicação que irá realizar a chamada de procedimento remoto, um servidor onde existe uma aplicação que irá processar o procedimento requisitado, assim como um serviço, o *stub* ou apêndice do cliente/servidor, que irá traduzir as mensagens entre cliente-servidor definidas pelo protocolo RPC e também realizar as invocações dos procedimentos, e para completar uma interface de rede para comunicação (BLOOMER, 1992).

O conceito por trás do RPC é que uma máquina A tem um de seus processos realizando uma chamada de procedimento em uma outra máquina B. Enquanto ocorre o processamento, o processo de A fica suspenso aguardando o retorno que deverá conter as informações que requisitou. As informações são transferidas do processo solicitante para o que irá computar pelo meio de parâmetros e voltam no resultado do procedimento. Durante a troca de mensagens o programador não vê o que esta acontecendo, esta técnica é conhecida como RPC (TANENBAUM; STEEN, 2007) .

O RPC tem como objetivo fazer com que a chamada remota se pareça o máximo possível com uma local, ou seja, ser transparente, e os procedimentos *send* (enviar) e *receive* (receber), métodos utilizados para a transferência de mensagens os são em seus acessos, o que é importante para sistemas distribuídos. Transparência aqui significa que o aplicativo não sabe que determinada chamada esta sendo realizada remotamente, e vice-versa (TANENBAUM; STEEN, 2007).

As mensagens de RPC são bens estruturadas e, portanto, não são apenas pacotes de dados. Cada mensagem é endereçada a um *daemon*[3] (*Disk And Execution MONitor* - *Monitor de Execução e de Disco*) RPC, o apêndice cliente/servidor ou *stub*, escutando uma porta do sistema remoto e contém um identificador da função a ser executada e os parâmetros que devem ser passados para essa função. Após a execução o retorno é enviado ao solicitante em uma mensagem separada (SILBERSCHATZ; GALVIN; GAGNE, 2010).

Uma porta é um número identificador inserido no início de um pacote de mensagem que faz com que haja diferenciação dos diversos serviços de rede que estão alocado para um endereço (SILBERSCHATZ; GALVIN; GAGNE, 2010) .

O apêndice de servidor/cliente, ou *stub* (termo em inglês), é o responsável por ocultar detalhes da comunicação entre cliente-servidor, assim como pela tradução de dados do formato XDR, caso este seja utilizado e que será tratado mais a frente. Pode existir um apêndice para cada procedimento existente, sendo necessário passar os parâmetros corretos para a invocação do mesmo no lado cliente, o *stub* do cliente então os organiza em uma mensagem própria para a rede, e localiza a porta correta para enviar ao *stub* do servidor que realiza a chamada no mesmo (SILBERSCHATZ; GALVIN; GAGNE, 2010).

Segundo Tanenbaum e Steen (2007) as etapas em uma chamada de procedimento remoto comum são:

[3] Um programa de computador que roda de forma independente em segundo plano, geralmente provendo um serviço.

1 Procedimento cliente chama o apêndice de cliente de maneira normal (chamada local).

2 Apêndice do cliente constrói a mensagem e chama o sistema operacional.

3 SO do cliente envia a mensagem para o SO remoto, ou de destino.

4 A mensagem é enviada ao apêndice do servidor pelo SO.

5 Apêndice do servidor realiza o desempacotamento da mensagem, obtendo os parâmetros e o procedimento a ser executado e realiza então a chamada do servidor.

6 O servidor processa o procedimento e retorna o resultado ao apêndice do servidor.

7 Resultado é empacotado pelo apêndice do servidor, que também chama o SO do servidor.

8 O SO do servidor envia a resposta para o SO do cliente.

9 A resposta é direcionada ao apêndice do cliente pelo SO cliente.

10 O apêndice do cliente desempacota o resultado e o envia ao procedimento cliente.

Na Figura 05 encontra-se uma representação do modelo simples de RPC com a utilização de porta fixa para as chamadas , enquanto na Figura 06 há a adição de um serviço *portmapper*. Nos diversos sistemas pode-se implementar o protocolo mapeador de portas - *portmapper*, junto ao protocolo de transporte para que assim a aplicação consiga obter portas dinamicamente para uso nas chamadas de RPC. Além disso, ele fica responsável por mapear as chamadas de procedimento remoto, as versões das chamadas e as portas específicas utilizadas, este tipo de aplicação é

chamada de *portmap* e geralmente fica à escuta de conexões nas portas TCP e UDP 111 (RIBEIRO, 2005).

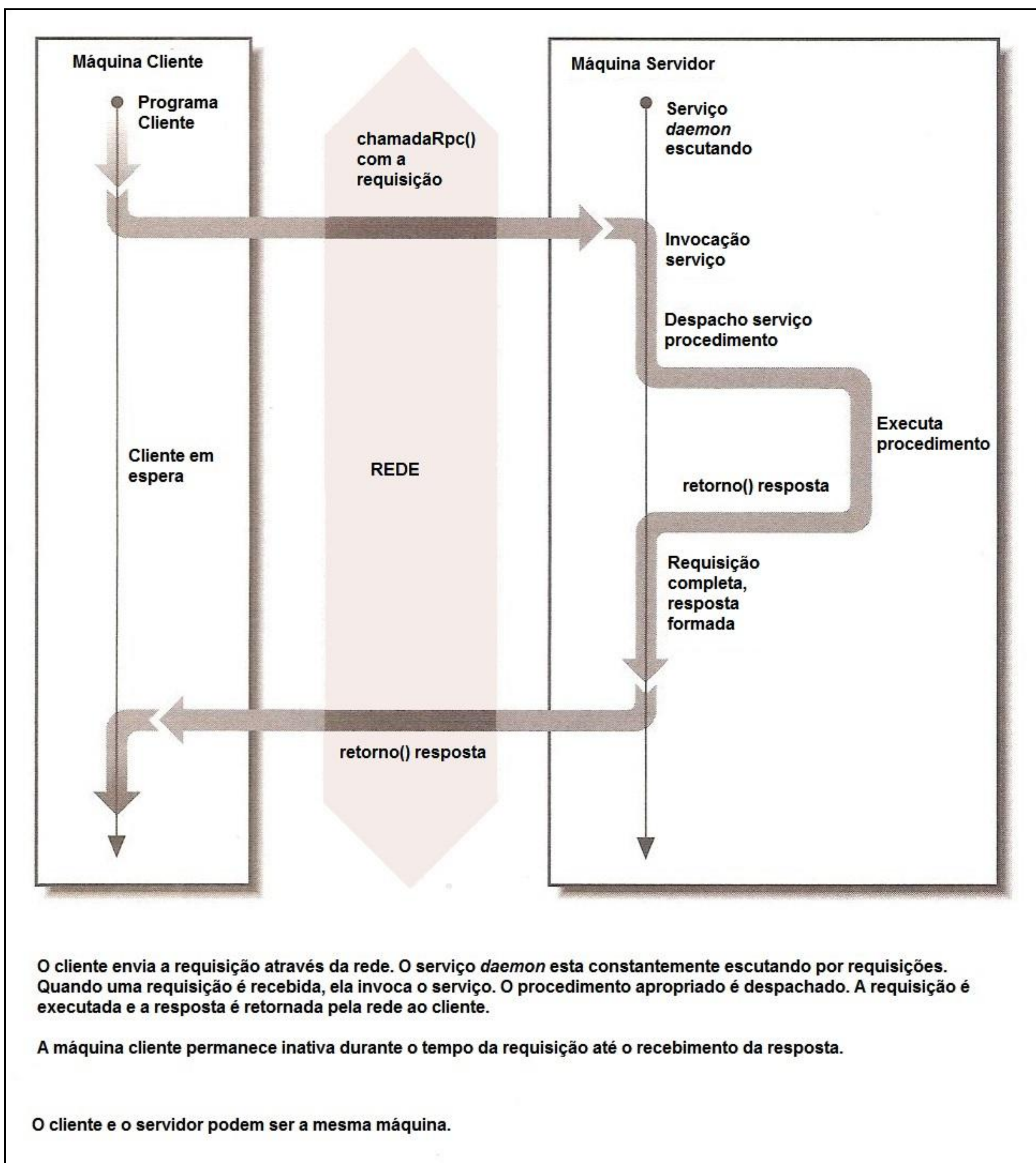


Figura 05 - Comunicação RPC convencional. **Fonte: Adaptado de Bloomer (1992, p.13).**

A identificação de cada procedimento remoto é única através do número do programa, número da versão e número do procedimento. Número de programa identifica o conjunto de procedimentos remotos correlacionados, cada um deles pode conter um de versão, a qual pode conter um ou mais procedimentos remotos,

cada tendo um único número também, isso será explicado com mais detalhes na sessão "3.2 Desenvolvimento com RPC" (RIBEIRO; 2005).

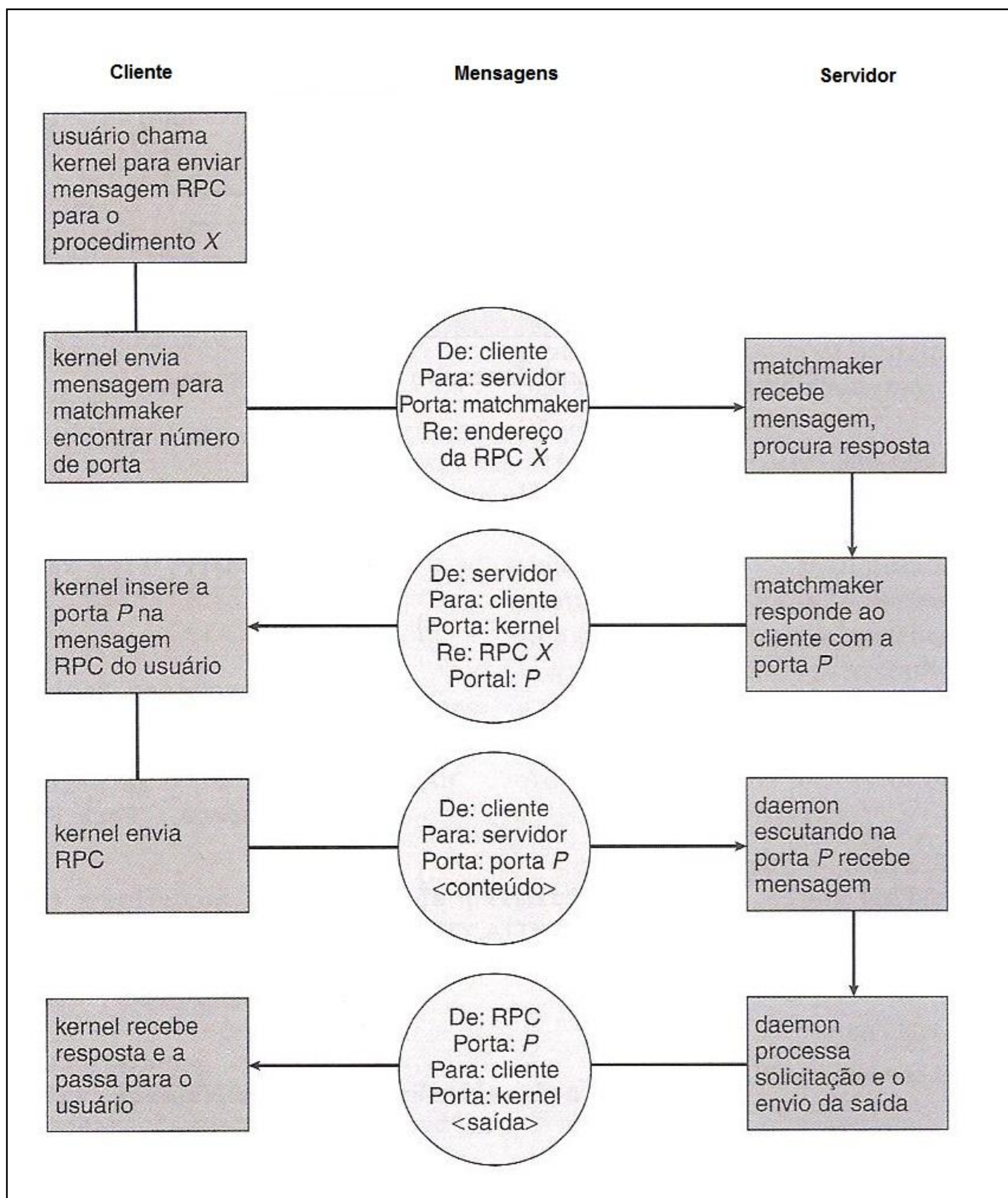


Figura 06 - Execução de uma chamada de procedimento remota. Fonte: Adaptado de Silberschatz e Galvin (2010, p.75).

Tanenbaum e Steen (2007) atentam para a ocorrência da correta troca de mensagens entre diferentes sistemas, os quais possuem diversas abordagens de representação de dados, como no caso do *big-endian* e *little-endian,* onde no

primeiro o byte significativo é armazenado primeiro enquanto no outro o menos significativo o é. Nestes casos é preciso que estes se utilizem do mesmo formato de mensagens e também de uma mesma representação de dados, o que resulta em ambos utilizarem o mesmo protocolo RPC. Caso contrário ao realizar chamadas entre sistemas distintos os resultados seriam variáveis devido a forma de interpretação de cada um, resultando claramente em erros.

Silberschatz, Galvin e Gagne (2010) complementam que para evitar problemas entre as diferentes arquiteturas e SOs utilizadas em um sistema distribuído, alguns sistemas RPC utilizam-se de uma representação independente dos dados conhecida como representação de dados externa, a XDR (*eXternal Data Representation - Representação de dados externa*), a qual no lado do cliente converte os dados para a arquitetura dela, e no lado do servidor realiza a sua transformação para o formato da máquina destino.

A XDR foi estendida e é a implementação da SUN em seu SUN RPC, ou ONC (*Open Network Computing - Rede aberta de computação*) RPC como também é conhecida, de uma IDL (*Interface Definition Language - Linguagem de definição de interface*) a qual é uma linguagem utilizada para definir as chamadas que podem ser realizadas assim como os tipos de dados que serão utilizados. Em cada protocolo RPC existente é possível ser utilizado uma IDL diferente, no caso da ONC RPC é a XDR, enquanto na NCS (*Network Computing System - Sistema computacional em rede*) RPC utiliza-se de NDR (*Network Data Representation - Representação de dados de rede*), enquanto no Microsoft RPC utiliza-se do MIDL (*Microsoft Interface Definition Language - Linguagem de definição de interface Microsoft*) que se baseia na NDR também, porém com o padrão do *Open Group Distributed Computing Environment (Grupo aberto de ambiente de computação distribuída)* (BLOOMER, 1992; MICROSOFT, 2013).

Diferentemente das chamadas locais as quais falham em situações extremas, as RPCs devido a utilização da rede, embora não necessariamente seja sempre assim já que pode existir chamadas para a mesma máquina, podem sofrer de erros de execução mais comuns e até mesmo de duplicação, em razão de problemas durante a transmissão. Uma forma de se precaver quanto a isso é implementar

chamadas com a opção de ser executada "somente uma vez" ao contrário de "no máximo uma vez", entretanto a complexidade da codificação pode ser elevada (SILBERSCHATZ; GALVIN; GAGNE, 2010).

Silberschatz, Galvin e Gagne (2008, 2010) propõe para a implementação de "no máximo uma vez" a utilização de carimbos com data e hora para cada mensagem enviada com um pedido de chamada, e o servidor então realizar o controle para não repetir a execução de algo já realizado. Para atingir o "somente uma vez" seria necessário utilizar-se do "no máximo uma vez" e acrescentar o uso de mensagens ACK (Acknowledge - Reconhecimento) que seriam responsáveis por avisar o cliente que a requisição foi processada, garantindo sempre a execução da chamada.

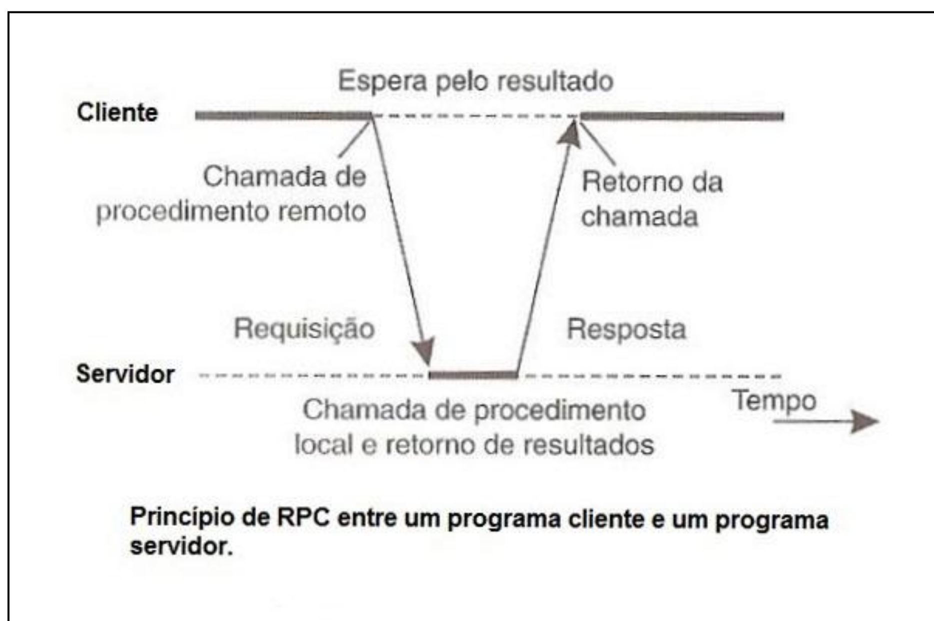


Figura 07 - Processo de chamada RPC convencional em relação ao tempo. Fonte: Adaptado de Tanenbaum e Steen (2007, pg. 77).

Existem alguns casos onde após a chamada de determinado procedimento em um servidor o cliente não precisa esperar pela resposta e pode seguir com outras funções sem a necessidade de permanecer bloqueado, como ocorre em chamadas convencionais (tanto locais, como remotas), até que receba uma reposta. Nestes casos é possível fazer com que o servidor de RPC envie imediatamente uma resposta para o cliente informando o recebimento da chamada e o desbloqueando para executar

outras atividades, este tipo é conhecido como **RPC assíncrona**. Exemplos de casos sem a necessidade de bloqueio são: processamento em lotes, requisição de gravação em arquivos ou banco de dados, inicialização de serviços remotos, entre outros (TANENBAUM; STEEN, 2007).

Pode-se encontrar variantes da RPC assíncrona como a RPC de uma via, onde o cliente somente envia a requisição e não espera nenhum reconhecimento por parte do servidor para saber se ela foi aceita ou não, o que pode ocasionar problemas caso não haja alta confiabilidade. Outra variante é a RPC assíncrona deferida, a qual o cliente realiza uma chamada e continua com o que esta fazendo, até que o servidor lhe envia outra RPC que contém os dados que foram requisitados (TANENBAUM; STEEN, 2007).

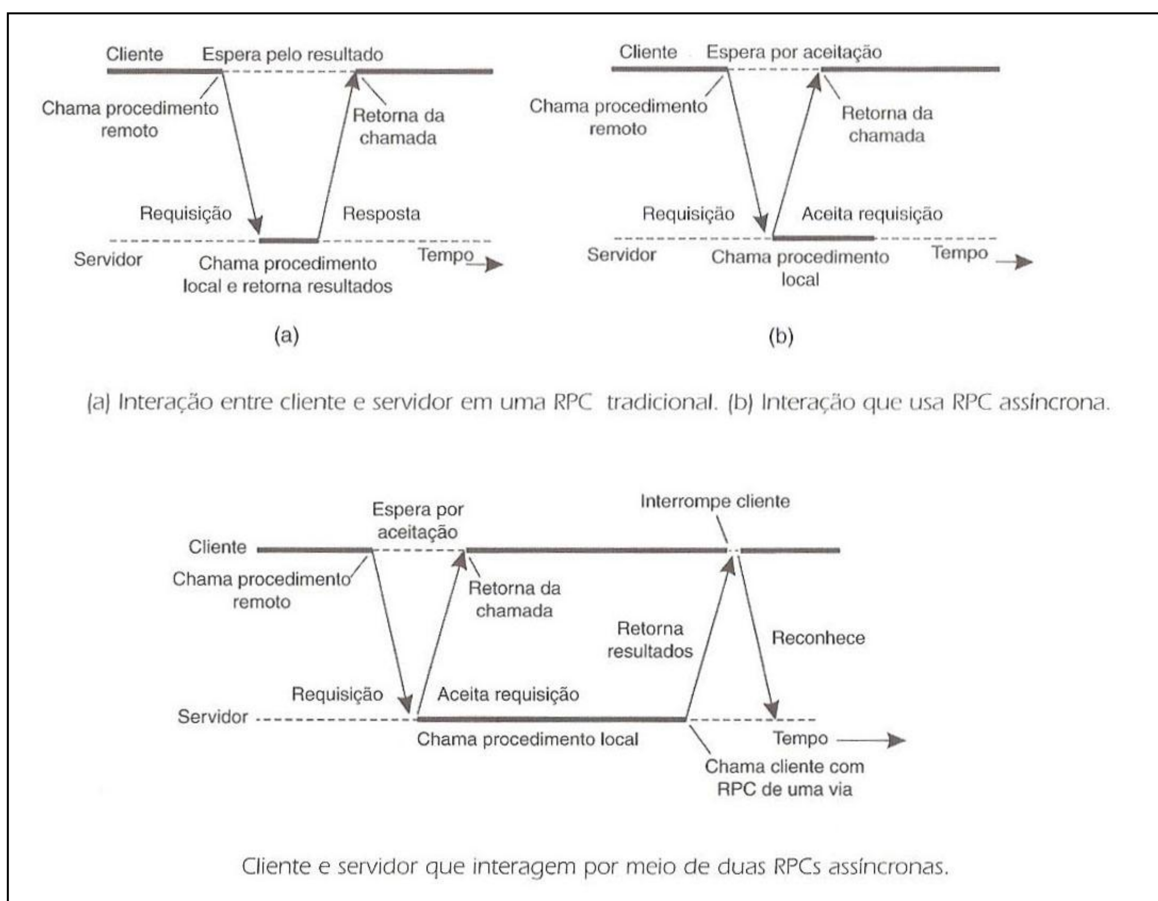


Figura 08 - Comparação entre RPC tradicional e RPC assíncrona. Fonte: Adaptado de Tanenbaum e Steen (2007, pg. 81).

Na Figura 07 está a relação tempo e processo em uma RPC convencional que é síncrona, já na Figura 08 no item "a" tem-se novamente a

RPC tradicional e no item "b" uma assíncrona de uma via, onde se pode observar que o processo cliente apenas fica suspenso enquanto espera a aceitação da requisição, e não durante todo o processamento. Por fim na parte inferior encontra-se uma assíncrona deferida, que em determinado ponto de sua execução será interrompida por uma nova chamada vinda do servidor, a qual conterá a mensagem com a resultado da sua última requisição.

3.2 DESENVOLVIMENTO COM RPC

Neste trabalho será utilizado como foco a implementação feita pela ONC RPC já que ela esta presente em grande parte dos sistemas Unix/Linux de hoje devido a sua utilização nos serviços de NIS (*Network Information Service - Serviço de Informação da Red*e) e NFS (*Network File System - Sistema de Arquivos de Rede*), além das bibliotecas para a execução de RPCs (as quais costumam vir junto com os SOs, inclusive no Windows, embora seja para o protocolo Microsoft RPC), ainda há alguns ambientes que já vêm com as ferramentas necessárias para o desenvolvimento com RPC (BLOOMER, 1992; MICROSOFT, 2013).

Ao desenvolver aplicações distribuídas com RPC detalhes referente a interface e subsistema de rede não precisam ser levados em conta, já que o protocolo permite a transmissão das mensagens de forma independente, separando os elementos lógicos e físicos envolvidos na comunicação, tornando o modelo cliente-servidor mais poderoso e fácil para programar. Porém é importante que a aplicação saiba qual protocolo de transporte esta sendo utilizando pelo RPC, já que no caso do TCP/IP não será necessário controlar problemas relacionados a transmissão de dados, porém já no UDP a situação muda, sendo preciso diversos controles para receber corretamente os dados transmitidos (RIBEIRO, 2005).

Segundo Ribeiro 2005 para o desenvolvimento de uma aplicação cliente servidor utilizando RPC, é preciso :

- Especificar o protocolo que será responsável pela comunicação entre cliente-servidor;

- Desenvolver o aplicativo que será provedor das funções, ou seja, o servidor.

- Desenvolver o aplicativo que fara as requisições, ou seja, o cliente.

- Gerar os arquivos necessários a compilação, como o arquivo da IDL por exemplo;

- Compilar a aplicação do servidor e a do cliente, de acordo com a arquitetura onde o aplicativo será executado.

3.2.1 DEFINIÇÃO DO PROTOCOLO

No protocolo é definido o formato das mensagens entre cliente e servidor, e nele deverá constar (RIBEIRO, 2005):

- Identificação do programa a ser chamado;

- Qual procedimento dentro do programa será chamado.

- Qual versão deve ser chamada.

- Os tipos de dados que serão utilizados como argumentos, ou parâmetros, na chamada.

- O tipo dos dados do retorno.

Com a utilização do ONC RPC o protocolo será escrito em XDR, já em outras implementações outras IDLs serão utilizadas, no Microsoft RPC por exemplo será utilizada a MIDL. Na Figura 09 tem-se um modelo de exemplo do protocolo ONC RPC (RIBEIRO, 2005; MICROSOFT, 2013).

Por convenção os nomes dos programas, versões e procedimentos devem ser escritos em letra maiúscula. As palavras "*program*" e "*version*" são reservadas, não podendo então serem utilizadas para programas, procedimentos ou versões. O nome de uma versão e o número não pode ser repetido para um mesmo nome de programa, pois isso inviabiliza a utilização de várias versões para um mesmo programa, a existência de mais de uma versão é utilizado quando se tem um desenvolvimento por etapas. Para um procedimento seu nome também não pode ser repetido em uma mesma versão (RIBEIRO, 2005).

```
/* Exemplo modelo para um Protocolo RPC no modelo ONC RPC */
program NOMEPROGRAMA { /* nome do programa que identifica o conjunto de procedimentos */
        version NOMEVERSAO { /* em um mesmo programa é possível existir várias versões */
                [tipoRetorno] NOMEPROCEDIMENTO1([tipoParâmetro] Parâmetro) = X /* X representa o
                                                                                  * número do procedimento
                                                                                  */
        [tipoRetorno] NOMEPROCEDIMENTO2([tipoParâmetro] Parâmetro) = X+1 /* não precisa ser X+1, porém
                                                                           * não pode ser igual a X ou outro
                                                                           * valor já utilizado.
                                                                           */
        } = Y; /* Y representa o número da versão*/
} = Z /* representa o número do programa, existe um intervalo de numeração recomendado
       * a ser utilizado e deve ser escrito na forma hexadecimal.
       */
```

Figura 09 - Modelo de protocolo RPC ONC RPC. Fonte: Adaptado de Ribeiro (2005, pg. 296).

Os tipos aceitos no protocolo ONC RPC são: *int*, *char*, *string*, *void*, *enum*, *bolean*, *float*, *doule* e *structure*. Já a numeração do programa deve ser escrito em hexadecimal e recomenda-se estar dentro de um intervalo especificado pela ONC RPC conforme a Figura 10 (RIBEIRO, 2005):

```
00000000 - 1fffffff - definido pelo rpc@sun.com
20000000 - 3fffffff - definido pelo usuário (aplicações internas devem estar neste intervalo)
40000000 - 5fffffff - transiente
60000000 - 7fffffff - reservado
80000000 - 9fffffff - reservado
a0000000 - bfffffff - reservado
c0000000 - dfffffff - reservado
e0000000 - ffffffff - reservado
```

Figura 10 - Intervalos de numeração do programa ONC RPC. Fonte: Adaptado de Ribeiro (2005, pg. 297).

No primeiro grupo da Figura 10 ficam os programas de uso geral que são padronizados em todos os sistemas e que ainda são de administração da SUN MICROSYSTENS, mesmo agora ela sendo subsidiária da Oracle. No segundo grupo ficam as aplicações específicas dos usuários. No ONC RPC nos sistemas Unix/Linux é possível encontrar o arquivo "*/etc/rpc"* o qual lista os serviços e suas respectivas portas que estão sendo utilizados no sistema (RIBEIRO, 2005).

```
/* Exemplo de um protocolo */

program PROGRAMAPING{
        /*
        * Última versão criada, ainda em testes
        */
        version PING_V2 {
                void PING_PROCEDIMENTO_NULL(void) = 1;
                int PING_PROCEDIMENTO_BACK(int NUM) = 2;
        } = 2;
        /*
        * Primeira versão, atualmente em produção.
        */
        version PING_V1{
                void PING_PROCEDIMENTO_NULL(void) = 1;
        } = 1;
} = 0x20000001;
```

Figura 11 - Exemplo de protocolo RPC ONC RPC. Fonte: Adaptado de Ribeiro (2005, pg. 298).

No exemplo da Figura 11 esta um programa chamado "PROGRAMAPING" o qual tem o número de identificação "0x2000001", nele existem duas versões, sendo a "1" a versão em produção de nome "PING_V1" contendo apenas um procedimento de numeração "1" chamado "PING_PROCEDIMENTO_NULL" o qual não tem parâmetros nem retorno.

A segunda versão chamada de "PING_V2" possui número "2" e possui dois procedimentos, o de número "1" é o mesmo da versão "1", já o de número "2" tem o nome "PING_PROCEDIMENTO_BACK", contendo como retorno um dado do tipo *int* (número inteiro) e como parâmetro nomeado "NUM" do tipo *int*. Por convenção o arquivo deverá ser salvo com a extensão ".x".

Na ONC RPC para se compilar o protocolo que foi definido utiliza-se um programa chamado RPCGEN, o qual está disponível no pacote de compiladores GLIBC, e como dito anteriormente alguns sistemas Unix/Linux já vem com ele por padrão, no entanto isso não é regra geral (RIBEIRO, 2005).

O RPCGEN irá ler o arquivo de protocolo "nomeprograma.x" e a partir dele irá gerar o apêndice ou *stub* do cliente e do servidor que terão nome de "nomeprograma_clnt.c" e "nomeprograma_svc.c", também irá produzir todos o filtros XDR necessários de acordo com os tipos de dados utilizados com nome de "nomeprograma_xdr.c" e um arquivo de cabeçalho "nomedoprograma.h" que deverá ser incluído no aplicativo do cliente e do servidor e em seus apêndices (BLOOMER, 1992).

Para criar os módulos do RPC, a forma mais simples é realizando a simples chamada do arquivo que contém o protocolo da aplicação com o RPCGEN, utilizando ainda o modelo do programa "PROGRAMAPING" tem-se então (considerar o nome do arquivo do programa como "ping.x") (RIBEIRO, 2005):

 # rpcgen ping.x

Serão então criados 3 arquivos: "ping.h", "ping_clnt.c" e "ping_svc.c". Sendo o primeiro o cabeçalho com a definições comuns ao servidor e cliente, o segundo o programa base com as chamadas de RPC para o cliente e o último as chamadas de RPC para o Servidor. Todos esses arquivos serão utilizados para compilar o aplicativo para o cliente e para o servidor e não devem ser alterados, arquivos completos estão disponibilizados no apêndice A (RIBEIRO, 2005).

Devido a utilização apenas de *void* e *int*, tipos de dados simples, não foi preciso a criação de um arquivo XDR, caso fosse necessário então o arquivo "ping_xdr.c" seria criado contendo os filtros XDR, os quais fazem a abstração para a comunicação entre arquiteturas diferentes (BLOOMER, 1992).

3.2.2 INTERFACE RPC

Com o protocolo definido é preciso agora que tanto a aplicação do servidor como do cliente sejam escritas de forma que respeitem os procedimento disponibilizados remotamente, em especial os tipos de dados que foram definidos no protocolo e a assinatura do procedimento. Será papel do servidor fazer o registro , isto é, conter os diversos procedimentos que poderão ser chamados pelo cliente,

assim como aceitar argumentos e retornar dados, do mesmo tipo que foram estabelecidos no protocolo (RIBEIRO, 2005).

De acordo com a necessidade de controle o RPC fornece diferentes níveis de interface para serem utilizados no desenvolvimento das aplicações cliente-servidor. Quanto mais controle ela oferecer mais código será necessário ser elaborado, muitas dessas funções são utilizadas pelo próprio RPCGEN (RIBEIRO, 2005).

Devido ao RPCGEN é possível criar aplicações somente com a sua utilização e chamando explicitamente no código a função *clnt_create* no programa principal do cliente. No entanto para elucidar as capacidades e potencial do RPC será feita uma visão geral das rotinas que podem ser utilizadas durante o desenvolvimento, em especial, de aplicações com maior controle e complexidade.

- Rotinas Simplificadas.

Permitem somente a utilização de um tipo de transporte durante a chamada do procedimento remoto, costumam ser as mais simples de implementar, nesta categoria encontram-se as funções *rpc_reg* e *rpc_call*, a primeira registra um procedimento como um programa RPC nos transportes especificados enquanto a segunda realiza uma chamada remota (RIBEIRO, 2005).

Exemplo pode ser observador na Figura 12 abaixo:

```
Função rpc_call e seus argumentos.
rpc_call(char *host, unsigned prognum, unsigned versum, u_log procnum, xdrproc_t
          inproc, char *in,xdrproc_t outproc, char *out, char *protocol).

rpc_call("endereco.com.br",0x3fffffffl, 0x1, 0x1, xdr_int, (char *)&inproc,
          xdr_int, (char *)&outproc, "UDP");
```

Figura 12 - Exemplo rotina simplificada *rpc_call*. Fonte: Adaptado de Ribeiro (2005, pg. 307).

- Rotinas de Alto Nível.

Diferentemente das simplificadas, as de alto nível precisam que haja uma negociação com o programa cliente antes de ser feita a chamada remota, e com o

programa servidor antes de receber as chamadas. Algumas funções neste grupo são (RIBEIRO, 2005):

- *clnt_create*: Cria uma entidade cliente em que se é informado o endereço do servidor, procedimento, versão e tipo de transporte a ser utilizado;

- *clnt_create_timed*: Similiar a anterior, porém é possível informar um tempo limite para que ocorra a negociação;

- *svc_create*: Manipula os protocolos de transporte especificados para o servidor;

- *clnt_call*: Realiza uma chamada remota em um servidor.

Exemplo pode ser visto na Figura 13 abaixo:

```
Função clnt_create e seus argumentos.
clnt_create(char *host, unsigned prognum, unsigned versnum, char *protocol)

clnt_create("192.168.1.1", NOMEPROGRAMA, NOMEVERSAO, "tcp")
```

Figura 13 - Exemplo rotina de alto nível *clnt_create*. Fonte: Adaptado de Ribeiro (2005, pg. 308).

Devido a utilização do RPCGEN é criado no cabeçalho ("nomeprograma.h") algumas definições que auxiliam o desenvolvimento como ao utilizar "NOMEPROGRAMA" e "NOMEVERSAO" serão palavras reservadas que irão representar o número do programa e o número da versão, respectivamente em hexadecimal.

- Rotinas intermediárias.

Com aumento da complexidade dos códigos desenvolvidos utilizando estas rotinas é possível atingir uma maior eficiência, em especial devido ao controle de detalhes na forma que serão realizadas as chamadas de procedimento remoto.

A função *clnt_tp_create* permite não só a criação de uma entidade cliente, assim como na *clnt_create*, mas além disso comporta a utilização de um transporte

específico. Já a *svc_tp_create_timed* manipula todos os protocolos de transporte especificados do lado do servidor conjuntamente com a possibilidade de especificar o tipo de transporte.

- Rotinas para experientes.

Neste grupo estão funções capazes de controlar os parâmetros do transporte a ser utilizado, entre elas estão *rpcb_set* que registra um endereçamento do servidor, enquanto a *rpcb_unset* faz o contrário. A *clnt_tli_create* permite a criação de uma entidade cliente ademais com especificação de parâmetros especiais do transporte, já a *svc_tli_create* é a correspondente para o servidor, criando uma entidade servidor.

3.2.3 ELABORAÇÃO DAS APLICAÇÕES CLIENTE E SERVIDOR.

Para desenvolver o código da aplicação cliente e servidor é possível partir de uma já existente e então adaptá-la para utilizar chamadas de procedimento remotas, assim como por outro lado é factível criar algo completamente novo do zero.

Uma calculadora que utiliza RPC, por exemplo, com as funções somar e subtrair precisa primeiramente ter seu protocolo definido, onde terá declarado os tipos de parâmetros e de retornos dos procedimentos, como já foi explicado anteriormente.

O protocolo para uma aplicação dessa seria conforme a Figura 14:

```
struct operando {
 int x;
 int y;
};

program CALCULADORA {
        version CALCULADORA_VERSAO {
                int SOMAR(operando) = 1;
                int SUBTRAIR(operando) = 2;
        } = 1;
} = 0x20000002;
```

Figura 14 - Protocolo Calculadora RPC. Fonte: Adaptado de Ribeiro (2005, pg. 312)

É importante ressaltar que no protocolo "calculadora.x" é utilizado a estrutura "operando" ao invés de informar dois parâmetros, por exemplo, na chamada de procedimento "SOMAR", a razão disso é a não possibilidade do uso de mais de um argumento durante a sua definição.

Gerando o código dos apêndices, biblioteca e definição de dados com o RPCGEN, tem-se os arquivos "calculadora.h", "calculadora_svc.c", "calculadora_clnt.c" e "calculadora_xdr.c". Nenhum desses arquivos devem ser alterados, porém o arquivo da biblioteca "calculadora.h" contém informações e definições importantes referente a implementação da codificação do servidor e cliente, conforme a Figura 15, sendo recomendável visualizá-lo.

```
// Definição das constantes de identificação do programa
#define CALCULADORA 0x20000002
#define CALCULADORA_VERSAO 1

// Definição das estruturas a serem utilizadas na aplicação
// devido ao protocolo para os parâmetros.|
struct operando {       int x;  int y;};
typedef struct operando operando;

// Assinatura das chamadas a serem utilizadas pelos clientes
#define SOMAR 1extern  int * somar_1(operando *, CLIENT *);
#define SUBTRAIR 2extern  int * subtrair_1(operando *, CLIENT *);

// Assinatura das chamadas a serem utilizadas no servidor.
extern  int * somar_1_svc(operando *, struct svc_req *);
extern  int * subtrair_1_svc(operando *, struct svc_req *);
```

Figura 15 - Dados importantes biblioteca Calculadora RPC. Fonte: Adaptado de Ribeiro (2005, pg. 313)

Ribeiro (2005) ainda explica que somente se o compilador suportar C++ é que serão definidas as assinaturas, portanto as chamadas dos procedimentos. Conforme pode ser observado na Figura 15 existem tanto os procedimentos a serem utilizados na aplicação cliente como na servidora, as quais por convenção são formadas da seguinte maneira *<nome definido no protocolo>_<número versão>_svc* para o servidor enquanto as para o cliente são no mesmo padrão porém sem *_svc* no fim.

Durante as chamadas elas irão receber como argumento a estutura *operando* que foi definida no início do protocolo, além disso as clientes recebem o argumento *CLIENT*, que é o ponteiro para a estrutura *CLIENT* que esta definida na biblioteca RPC "rpc/clnt.h" (RIBEIRO, 2005).

```
// Aplicação Cliente
#include <stdio.h>
#include <stdlib.h>
#include "calculadora.h" /*Cabeçalho gerado pelo RPCGEN contendo definições e interfaces*/

int main ( int argc, char *argv[])
{

        CLIENT *clnt;
        int x,y;
        if (argc!=4)
        {
                fprintf(stderr,"Uso: %s servidor num1 num2\n",argv[0]);
                exit(0);
        }

        clnt = clnt_create(argv[1], CALCULADORA, CALCULADORA_VERSAO, "tcp"); /* Chamada de interface de alto
                                                                              *  nível para criação do ponteiro
                                                                              */ CLIENT
        operando ops;
        ops.x = atoi(argv[2]);
        ops.y = atoi(argv[3]);
        printf("%d + %d = %d\n", ops.x, ops.y, *somar_1(&ops,clnt)); /* Chamada do procedimento Somar do cliente*/
        printf("%d - %d = %d\n", ops.x, ops.y, *subtrair_1(&ops,clnt)); /* Chamada do procedimento Subtrair do
                                                                          *  Cliente
                                                                          */
        return(0);
}


// Aplicação Servidor
#include "calculadora.h"

int * somar_1_svc(operando *argp, struct svc_req *rqstp) /* Chamada do procedimento Somar no Servidor */
{
        static int result;
        result = argp->x + argp->y;
        return &result;
}

int * subtrair_1_svc(operando *argp, struct svc_req *rqstp) /* Chamada do procedimento Subtrair no Servidor*/
{
        static int result;
        result = argp->x - argp->y;
        return &result;
}
```

Figura 16 - Exemplo para aplicação Calculadora RPC Cliente e Servidor. Fonte: Adaptado de Ribeiro (2005, pg. 317 e 319)

Com o protocolo definido e as interfaces necessárias geradas deve-se produzir o código da aplicação servidora e cliente como próximo passo, na Figura 16 tem-se um exemplo simplificado de como isso pode ser feito.

Neste ponto é importante observar algumas diferenças existentes entre chamadas locais e remotas conforme Ribeiro (2005) aponta:

- Na função remota é utilizado ponteiros para os argumentos e não os próprios argumentos.

- Como dito anteriormente, as chamadas de procedimento remoto recebem apenas um parâmetro, enquanto as locais podem receber vários. É possível através da opção "-N" no RPCGEN utilizar mais de um parâmetro, porém não é recomendado.

- O retorno das funções é feito através de um ponteiro e não do próprio valor e devem ser declarados com o modificador *static*.

Devido a geração do código através do RPCGEN, aliado a utilização do *portmapper* assim que o programa servidor for executado, será realizado automaticamente como um procedimento disponível somente esperando ser requisitado. No caso do programa cliente, neste exemplo, é utilizado o comando *clnt_create* para criar a conexão com o programa remoto, é nesta função que são informados o servidor, o programa, versão e protocolo necessários para a comunicação (RIBEIRO, 2005).

Ainda no programa cliente, para efetuar a chamada remota do procedimento SOMAR ou SUBTRAIR, será preciso declarar uma estrutura do tipo operando, no exemplo *ops,* a qual irá receber os valores passados como parâmetros na chamada do programa. Por fim a estrutura *ops* e o ponteiro *CLIENT* obtido através da função *clnt_create* serão passados como parâmetros para os procedimentos. O resultado obtido deverá ser lido como um ponteiro também (RIBEIRO, 2005).

Com todos os arquivos codificados é preciso agora compilá-los, algumas delas não exigem ordem alguma, no entanto a compilação dos programas finais, cliente e servidor, só podem ser realizadas quando os itens relacionados já estiverem compilados. Uma possível ordem, desde o primeiro processo, utilizando o *GCC* (*GNU Compiler Collection - Coleção de Compiladores GNU*) seria conforme a Figura 17 demonstra.

```
/* Gerar arquivos de biblioteca, apêndices e definição de dados */
# rpcgen calculadora.x
Resultado: Criação dos arquivos "calculadora_svc.c",
        calculadora_clnt.c", "calculadora.h" e "calculadora_xdr.c"

/* Formar programa objeto do cliente */
# cc -c -o calculadora_cliente.o -g calculadora_cliente.c
Resultado: Criação do arquivo "calculadora_cliente.o"

/* Formar programa objeto do apêndice do cliente */
# cc -c calculadora_clnt.c
Resultado: Criação do arquivo "calculadora_clnt.o"

/* Formar programa objeto do XDR */
# cc -c calculadora_xdr.c
Resultado: Criação do arquivo "calculadora_xdr.o"

/* Construir o executável do cliente */
# cc - o calculadora_cliente calculadora_cliente.o calculadora_clnt.o
        calculadora_xdr.o
Resultado: Criação do aplicativo calculadora_cliente

/* Formar programa objeto do servidor */
# cc -c -o calculadora_servidor.o calculadora_servidor.c
Resultado: Criação do arquivo "calculadora_servidor.o"

/* Formar programa objeto do apêndice do cliente */
# cc -c calculadora_svc.c
Resultado: Criação do arquivo "calculadora_svc.o"

/* Construir o executável do servidor */
# cc -o calculadora_servidor calculadora_servidor.o calculadora_svc.o
        calculadora_xdr.o
Resultado: Criação do aplicativo calculadora_servidor
```

Figura 17 - Compilando aplicação Calculadora RPC Cliente e Servidor. Fonte: Adaptado de Bloomer (1992, pg. 9 e 10)

Para a execução dos aplicativos RPC é preciso que o *portmap* esteja em execução, na implementação ONC RPC existe um programa *portmap* como serviço *portmapper*, para iniciá-lo é preciso entrar com o comando:

\# *rpc.portmap*

Em sistemas mais novos é possível que haja no lugar do *portmap* o *rpcbind*, o qual executa a mesma função (é um *portmapper*) e também deve ser iniciado antes:

\# *rpcbind*

Para verificar os programas que estão com portas registradas na máquina basta utilizar o comando *rpcinfo*, o qual vem junto na GLIBC e além disso exibe o número do programa, as versões disponíveis para chamada e os protocolos que aceita comunicação.

É importante ressaltar a verificação das portas permitidas no *firewall*[4], caso este esteja ativo, pois com a utilização da comunicação pela rede se houver bloqueio das portas as aplicações não irão se comunicar.

Com o ambiente e os aplicativos prontos é possível iniciar a execução dos mesmo, conforme demonstra a Figura 18 :

```
/* Iniciando aplicação servidor */
Máquina Servidora
# ./calculadora_servidor &
/* Utilização do & significa iniciar execução em segundo plano*/

/* Executando aplicação cliente */
Máquina Cliente
# ./calculadora_cliente 192.168.1.1 10 10
/* IP da Máquina Servidora é 192.168.1.1 */
Resposta escrita na tela do cliente:
# 10 + 10 = 20
# 10 - 10 = 0
```

Figura 18 - Executando as aplicações Calculadora RPC. **Fonte: Criada pelo autor.**

Os exemplos apresentados foram feitos com a utilização da implementação ONC RPC da SUN, no entanto existem outros distribuidores com suas versões do protocolo, como a DISTINCT[5] com foco em RPC para sistemas Windows e podendo utilizar-se das linguagens C/C++, JAVA e C#.net para o desenvolvimento, assim como outros projetos Open source (Código aberto) ou não realizados por diversos grupos como o já citado DCE e o Windows RPC, tendo inclusive universidades envolvidas no oferecimento de ferramentas (RIBEIRO, 2005).

---

[4] Software que policia o fluxo de entrada e saída de dados da rede de um computador.
[5] Disponível em: < http://www.onc-rpc-xdr.com > Acesso em: 01/06/2013

4 ESTUDO DE CASO

O estudo de caso que será apresentado neste trabalho tem como intuito avaliar se existem impactos significativos no desempenho de uma aplicação ao executá-la através de chamadas de procedimento remotas, RPCs, em relação ao seu processamento convencional, o sequencial local, visando assim avaliar a viabilidade das aplicações RPC no quesito desempenho.

Para a realização deste estudo foram utilizadas duas máquinas virtuais criadas no Oracle VirtualBox[6] conectadas através de uma rede *Ethernet* 1 Gbps (Gigabits[7] por segundo) no modo *Bridge*[8]. Ambas estiveram instaladas com a distribuição Linux CentOS 6.3 32-bits a qual se utiliza do Kernel[9] na versão 2.6.32-279, com 512 MBs (Megabytes) de memória. Cada máquina tem "virtualizado" apenas 1 CPU, sendo a referência da máquina hospedeira um Intel i7 2600k funcionando a 4.5 Ghz.

Foi utilizado um algoritmo *quick sort* (ordenação rápida) adaptado para que na base de seu funcionamento sejam criados "*x*" valores aleatórios os quais estes serão ordenados "*n*" vezes de acordo com os parâmetros informados, além de calcular o tempo total da execução do mesmo em segundos. No algoritmo sequencial o cálculo do tempo se inicia logo após a declaração de variáveis e termina em seguida da última mensagem. Já no algoritmo com RPC existem dois tempos, o primeiro relativo a execução do *quick sort* e o segundo ao tempo total, este desde a declaração das variáveis até o retorno da chamada de procedimento remoto e fim do aplicativo cliente.

O mesmo algoritmo foi adaptado para um conjunto de aplicação RPC em que o servidor será o responsável por realizar o processamento e calcular o tempo, enquanto o cliente será responsável pela passagem de parâmetros e realização da chamada de procedimento remoto.

---

[6] Aplicativo utilizado para simular máquinas reais de forma virtual.
[7] Unidade de armazenamento de informações em função do tempo.
[8] Bridge (Ponte): Neste modo as máquinas virtuais interceptam e injetam os dados na placa do computador hospedeiro tornando-se um adaptador.
[9] Núcleo do sistema operacional, principal responsável pelas interações software e hardware, entre outros.

O algoritmo *quick sort* desenvolvido tem como fundamento o descrito por Cormen et al. (2002) o qual apóia-se ao paradigma de dividir e conquistar[10] e possui uma boa eficiência média em relação ao pior caso (o mais lento) e o melhor (mais rápido).

Os códigos completos desenvolvidos para este estudo estão disponíveis no apêndice A para visualização e até mesmo implementação, assim como os exemplos deste trabalho eles também foram criados na linguagem de programação C utilizando ONC RPC, o qual vem com ambiente preparado para a codificação na distribuição Linux CentOS 6.3.

O processo do estudo foi baseado em três casos, o primeiro caso é a execução sequencial, o segundo a execução através de RPC porém na mesma máquina, e o terceiro e último caso a execução RPC com uma máquina servidora e outra cliente.

Cada algoritmo foi executado 5 vezes com os parâmetros de 5.000.000 dados (números aleatórios) a serem criados e ordenados, além da realização de 20 repetições desse processo (cada repetição gera um novo conjunto de dados) e então o valor médio do tempo de execução total foi calculado e considerado. Nos algoritmos também é encontrado o cálculo de tempo máximo, médio e mínimo, relacionado a cada ordenação, no entanto apenas foi analisado o tempo médio para garantir que durante as execuções não houve uma média superior ou inferior ao comum devido a situação a qual vários piores casos aconteçam em sequencia, no apêndice B é encontrado imagens da execução do estudo de caso.

Durante a execução somente as duas máquinas virtuais estiveram em execução para evitar possíveis cargas de processamento que comprometam os resultados.

Os dados obtidos serão apresentados por tabelas sendo que:

- Teste nº : Representa a ordem em que foi realizada o teste.

---

[10] Técnica na qual primeiramente os dados são particionados em dois grupos e então ordenados

- Tempo médio das ordenações: Tempo médio que cada ordenação completa levou para ser efetuada.

- Tempo total para todas as ordenações: Tempo total que as 20 ordenações levaram para serem realizadas.

- Tempo total da aplicação: Tempo total para a execução de toda a aplicação, no caso do *quick sort* sequencial representa o tempo total para todas as ordenações, já no *quick sort* distribuído (com RPC) será o tempo da aplicação cliente realizar a chamada, somado ao tempo de processo do algoritmo no servidor e o retorno e apresentação do resultado.

4.1 PRIMEIRO CASO - EXECUÇÂO SEQUENCIAL

O primeiro caso será utilizado como controle delimitando o desempenho padrão esperado, para isto será utilizado a execução convencional que é a sequencial em uma das máquinas virtuais.

Os dados obtidos estão ilustrados na Tabela 1 abaixo:

Tabela 1 - Dados primeiro caso. Execução sequencial, uma máquina.

Teste nº	Tempo médio das ordenações.	Tempo total para todas ordenações	Tempo total da aplicação
1	0,46 segundos	10,57 segundos	10,57 segundos
2	0,46 segundos	10,55 segundos	10,55 segundos
3	0,46 segundos	10,56 segundos	10,56 segundos
4	0,46 segundos	10,57 segundos	10,57 segundos
5	0,46 segundos	10,55 segundos	10,55 segundos
Média	**0,46 segundos**	**10,56 segundos**	**10,56 segundos**

Fonte: Criada pelo autor.

O algoritmo executado se encontra no Apêndice A com o nome de "quickSortSeq" e seguiu o padrão descrito anteriormente para a sua aplicação.

---

através de chamadas recursivas.

4.2 SEGUNDO CASO - EXECUÇÃO RPC MESMA MÁQUINA

O segundo caso será realizado para avaliar a diferença entre o processamento sequencial convencional e o sequencial distribuído dentro de um mesmo ambiente computacional, ou seja, computador.

Os dados obtidos estão ilustrados na Tabela 2 abaixo:

Tabela 2 - Dados segundo caso. Execução distribuída sequencial na mesma plataforma.

Teste nº	Tempo médio das ordenações.	Tempo total para todas ordenações	Tempo total da aplicação
1	0,45 segundos	10,40 segundos	10,41 segundos
2	0,45 segundos	10,41 segundos	10,42 segundos
3	0,46 segundos	10,37 segundos	10,39 segundos
4	0,46 segundos	10,45 segundos	10,45 segundos
5	0,46 segundos	10,44 segundos	10,46 segundos
Média	**0,46 segundos**	**10,41 segundos**	**10,43 segundos**

Fonte: Criada pelo autor.

Neste caso foram executados dois algoritmos, o primeiro foi o algoritmo do servidor, para estabelecer o serviço que irá receber e processar a chamada de procedimento remoto, sua codificação se encontra no Apêndice A com o nome de "quickSortRPC_SV". O outro que foi utilizado foi o "quickSortRPC" que representa a aplicação cliente que faz o pedido de processamento ao servidor, a chamada remota, e aguarda o resultado.

4.3 TERCEIRO CASO - EXECUÇÃO RPC MÁQUINAS DISTINTAS

Este caso representa a situação padrão em que é esperado a utilização de um aplicativo RPC, a execução em ambiente distribuído conectado através de uma rede, neste caso o sistema distribuído foi composto pelas duas máquinas virtuais, sendo uma a servidora do serviço, no caso a realização do *quick sort*, e a outra cliente efetuando a chamada remota e aguardando pelo resultado.

Os dados obtidos estão ilustrados na Tabela 3 abaixo:

Tabela 3 - Dados terceiro caso. Execução distribuída sequencial em máquinas distintas.

Teste nº	**Tempo médio das ordenações.**	**Tempo total para todas ordenações**	**Tempo total da aplicação**
1	0,46 segundos	10,43 segundos	10,45 segundos
2	0,46 segundos	10,45 segundos	10,46 segundos
3	0,45 segundos	10,38 segundos	10,40 segundos
4	0,46 segundos	10,41 segundos	10,43 segundos
5	0,45 segundos	10,40 segundos	10,41 segundos
Média	**0,46 segundos**	**10,41 segundos**	**10,43 segundos**

Fonte: Criada pelo autor.

5 DISCUSSÃO DOS RESULTADOS

A análise dos dados obtidos através do estudo realizado diferem do esperado, pois o tempo médio do segundo e terceiro caso foi menor que do primeiro, afinal nas chamadas locais não é preciso contatar o servidor, o que deveria poupar tempo de execução, em especial em relação ao último caso os quais o acesso via rede é mais necessário e mais intenso, que nos outros, o qual é mais lento que os meios de comunicações internas do computador.

É possível observar que o tempo de processo do algoritmo *quick sort* foi reduzido no segundo e terceiro caso , isso pode ter acontecido devido a adaptação do mesmo para a chamada remota, que pode tê-lo tornado mais eficiente, outra opção seria o fato da utilização de passagem de parâmetros por referência e não valor, por fim também tem-se a possibilidade que o algoritmo ficou com maior prioridade na implementação RPC para seu processo.

Em ADSC (Análise de Desempenho de Sistemas Computacionais) é de grande importância levar em consideração os fatores[11] que podem ter influência no resultado final, neste estudo um fator principal se dá ao ambiente virtual das máquinas em que esse foi realizado, o qual provavelmente minimizou a latência[12] e não prejudicou a aplicação em RPC em relação a convencional como era esperado, além de a rede que foi utilizada estar totalmente dedicada as aplicações não havendo tráfego de dados de outras fontes se tornando um ambiente próximo ao ideal, o qual não necessariamente irá ser refletido em uma ambiente real.

No entanto ao comparar-se com valores significativos, tem-se que o segundo e terceiro caso foram apenas 1% mais rápido que o primeiro, considerando uma possível margem de erro de 5% (ou até mesmo de 1%), pode-se portanto considerar que ambos tem o desempenho similar neste estudo.

---

[11] Elementos que influênciam o desempenho e funcionamento, no entanto não temos controle.
[12] Tempo de resposta, ou a diferença de tempo necessária entre uma requisição e a sua resposta, é afetada pela distância entre os elementos.

6 CONSIDERAÇÕES FINAIS

Neste trabalho inicialmente foi mencionado as principais áreas onde os sistemas distribuídos são empregados e a sua importância para a viabilidade das atividades delas.

No primeiro capítulo bibliográfico, seção 2 deste texto, foi feito o levantamento da arquitetura existente hoje que molda essas plataformas, desde as mais simples como o paralelismo no chip até as mais complexas como as nuvens computacionais, demonstrando suas diferentes características atingindo então a elucidação de sua existência, assim como também daquelas que estão em destaque atualmente devido a sua importância atual e potencial futuro, juntamente dos principais usos e dos desafios enfrentados na manutenção, evolução e disponibilização de tais estruturas.

O segundo capítulo bibliográfico, seção 3, foi apresentada e estudada uma das formas existentes para poder utilizar-se dos sistemas distribuídos, o RPC, expondo o seu funcionamento, suas capacidades, assim como suas particularidades e similaridades com as de sistemas computacionais convencionais.

Em seguida nesta mesma divisão foi visto e demonstrado como aplicar esta técnica, assim como o que já existe e a utiliza, foram elaborados exemplos de codificação e de emprego de ferramentas auxiliares no desenvolvimento de aplicações distribuídas como o RPCGEN e o *portmapper*. Também foram mencionadas outras implementações da mesma técnica que possuem outras abordagens, deixando aberto outras linhas de estudo.

Com isso o objetivo geral de se familiarizar com os sistemas distribuídos e com o desenvolvimento utilizando RPC para os mesmos foi atingido, e serviu de base para preparar a seção 4, o estudo de caso.

Durante o estudo de caso, foram elaboradas duas aplicações, sendo uma delas convencional e outra distribuída com o uso de RPC, em um ambiente Linux com ONC RPC, o qual foi "virtualizado" em duas máquinas distintas e então realizada a execução de ambos os programas para a comparação de desempenho

entre elas e assim alcançando os objetivos específicos, desenvolver uma aplicação distribuída com RPC e realizar uma comparação de desempenho entre ela e uma convencional (execução sequencial local).

Em consideração a ADSC devido a utilização do ambiente com máquinas virtuais, um fator que pode ter sido crucial aos resultados obtidos, fica em aberto a realização de trabalhos futuros, primeiramente com a verificação em um ambiente computacional distribuído conectado por uma rede local mas com máquinas físicas idênticas, porém separadas e mais adiante a análise quando estas estiverem interligadas pela Internet.

Ainda com relação aos estudos futuros, tem-se a possibilidade do estudo das vantagens e desvantagens entre a migração de dados de um banco de dados (enviar todos os dados, ou o arquivo todo, para outro lugar que então irá processá-lo(s)) e a migração da computação dele (transferência da computação, o caso convencional do RPC, onde se transfere funções e procedimento para outros lugares e que irão realizar o processo).

Foi observado neste estudo de caso que as aplicações RPC possuem desempenho similar a aplicações convencionais, no entanto fatores como o tráfico da rede, distância entre as máquinas foram minimizados devido ao ambiente utilizado, sendo plausível esperar que em um ambiente real haja alguma perda de desempenho.

Por fim mesmo que haja alguma degradação de desempenho, a proposta RPC para desenvolvimento distribuído, abrangendo diversas arquiteturas e SOs é muito poderosa e valiosa para as cada vez mais heterogêneas plataformas computacionais como os *clusters*, grades e nuvens os representantes dos sistemas distribuídos de alto nível, além da facilidade de não precisar lidar diretamente com interfaces de baixo nível da rede, porém para aplicações mais complexas elas estão disponíveis para aqueles que desejam maior controle.

7 REFERÊNCIAS

BLOOMER, John. ***Power programming with RPC***. [s. L.]: O'reilly Media, Inc., 1992. p. xxv-xxvi,1-35.

CITRIX SYSTEMS INC. ***The top 5 truths behind what the cloud is not:*** *separating the noise of what cloud is and what it's not*. Disponível em: <http://www.citrix.com/wsdm/restServe/skb/attachments/RDY7415/The%20top%205%20truths% 20behind%20what%20the%20cloud%20is%20not.pdf>. Acesso em: 01 dez. 2012.

CORMEN, Thomas H. et al. **Algoritmos**: teoria e prática. 2. ed. Rio de Janeiro: Elsevier, 2002. p. 117-120.

FOSTER, Ian; KESSELMAN, Carl; TUECKE, Steven. *The anatomy of the Grid: enabling scalable virtual organizations.* ***The international journal of high performance computing applications***, [s. L.], v. 15, n. 3, p.200-222, Fall 2001. Trimestral.Disponível em: <http://hpc.sagepub.com/content/15/3/200.full.pdf+html>. Acesso em: 28 jan. 2013.

MICROSOFT (Estados Unidos*).* ***Microsoft Interface Definition Language***. Disponível em: <http://msdn.microsoft.com/en-us/library/windows/desktop/aa367091%28v=vs.85%29.aspx>. Acesso em: 01 jun. 2013.

RIBEIRO, Uirá. **Sistemas distribuídos:** desenvolvendo aplicações de alta performance no LINUX. Rio de Janeiro: Axcel Books, 2005. p. 291 - 339.

SADASHIV, Naidila; KUMAR, S. M Dilip*. Cluster, Grid and Cloud Computing: a detailed comparison*. In: *INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE & EDUCATION*, 6., 2011, *Singapore.* ***International conference on computer science & education***. [s. L.]: *International conference on computer science & education*, 2011. p. 477 - 482.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Fundamentos de sistemas operacionais**. Tradução de Aldir José Coelho Corrêa da Silva. 8. ed. Rio de Janeiro: LTC, 2010. p. 73-75.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Sistemas operacionais com Java**. Tradução de Daniel Vieira. 7. ed. Rio de Janeiro: Elsevier, 2008. p. 85 - 87.

SILVA, Edna Lúcia da; MENEZES, Estera Muszkat. **Metodologia da pesquisa e elaboração de dissertação**. 3. ed. rev. atual. Florianópolis: Laboratório de Ensino a Distância da UFSC,2001. p. 21-22.

STERLING, Thomas. **Beowulf cluster computing with linux**. [S.l.]: The MIT Press,2001. p. 6-8. (Scientific and Engineering Computation).

SUN MICROSYSTEMS INC. ***Take your business to a higher level.*** Disponível em: <http://www.aeiseguridad.es/descargas/categoria6/4612546.pdf>. Acesso em: 20 dez. 2012.

TANENBAUM, Andrew S. **Redes de computadores**. Tradução: Vanderberg D. de Souza. 4 ed. Rio de Janeiro: Elsevier, 2003. p. 2-3.

TANENBAUM, Andrew S.**Organização estruturada de computadores**. Tradução de Arlete Simille Marques. 5. ed. São Paulo: Prentice Hall, 2007. p.322 - 382.

TANENBAUM, Andrew S; STEEN, Maarten Van.**Sistemas distribuídos**: princípios e paradigmas. Tradução de Arlete Simille Marques. 2. ed. São Paulo: Pearson Prentice Hall, 2007. p. 1, 75 - 84.

APÊNDICE A - Algoritmos do estudo de caso.

- Código fonte principal para o aplicativo cliente "quickSortRPC"

```
#include "qsrpc.h"
#include <time.h>
#include<sys/time.h>

int main(int argc, char *argv[])
{
    struct timeval progRpcIni, progRpcFim;
    CLIENT *cl;
    double *retorno, segs, microsegs, progRpcTotal;

    gettimeofday(&progRpcIni, NULL);

  if ((argc !=4) || (!isdigit(argv[2][0])) || (!isdigit(argv[3][0]))){
    fprintf(stderr, "Uso: %s servidorRPC qtdeDados qtdeRepeticoes\n", argv[0]);
    exit(1);
  }

    cl = clnt_create(argv[1], QUICKSORT_RPC, QSRPC_V1, "tcp");

    if (cl == NULL) {
        printf("erro: Nao foi possivel conectar ao servidor.\n \
        Recomendacoes:\n\
        - Verificar se o servidor esta no ar\n\
        - Verificar se a aplicacao do servidor esta rodando\n\
        - Verificar Firewall\n\
        - Verificar conexao entre as maquinas\n");
        return 1;
    }

    printf("Chamada requisitada\n");

    params pr;
    pr.qtdeDados = atoi(argv[2]);
    pr.repeticoes = atoi(argv[3]);

    retorno = quicksort_1(&pr,cl);
    printf("\n\nTérmino contagem programa quicksort. Tempo total %5.2lf segundos\n", *retorno);
    printf("Chamada finalizada\n");

    gettimeofday(&progRpcFim, NULL);

    segs = (double) progRpcFim.tv_sec - progRpcIni.tv_sec;
    microsegs = (double) progRpcFim.tv_usec - progRpcIni.tv_usec;
    progRpcTotal = (double) segs + microsegs/1000000;
    printf("\n\nTérmino contagem programa RPC. Tempo %5.2lf segundos\n",progRpcTotal);
    return 0;
}
```

Arquivo "qsrpc.c", compilado com o "qsrpc_clnt.c" e "qsrpc_xdr.c" gera o aplicativo equivalente ao "quickSortRPC"

- Código fonte principal para o aplicativo servidor "quickSortRPC_SV"

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "qsrpc.h"

    void qs(short int tabela[], int ini, int fim)
    {
        int i, j;
        int pivot;
        short int y;

        i = ini;
        j = fim;
        pivot = tabela[(ini+fim)/2];
        do
        {
            while((tabela[i]<pivot) && (i<fim))
            {
                i++;
            }

            while((pivot<tabela[j]) && (j>ini))
            {
                j--;
            }
            if(i<=j)
            {
                y = tabela[i];
                tabela[i] = tabela[j];
                tabela[j] = y;
                i++;
                j--;
            }
        }
        while(i<=j);

        if(ini<j)
        {
            qs(tabela, ini, j);
        }

        if(i<fim)
        {
            qs(tabela, i, fim);
        }
    }
```

Arquivo "qsrpc_servidor.c" (parte 1/3), ao ser compilado com o "qsrpc_svc.c" e "qsrpc_xdr.c" gera o aplicativo equivalente ao "quickSortRPC_SV".

```
    void quickSort(short int tabela[], int tamanho)
    {
        qs(tabela, 0, tamanho-1);
    }

    // Preenche a tabela com dados aleatórios
    void preenche(short int tabela[], int qtde)
    {
        int i;
        for(i = 0; i < qtde; i++)
        {
            tabela[i] = rand()%10000;
        }
    }


double * quicksort_1_svc (params *argp, struct svc_req *rqstp){

    static double retorno;

    int vezes = 0, primeiraVez = 1;
    int i;
    clock_t inicio, fim, progini, progfim;
    double diferenca, maior, menor, media, soma = 0;

    /*Inicio contagem tempo total do aplicativo*/
    progini = clock();
    printf("\nInício contagem tempo total do aplicativo quicksort");


    int tamanho = argp->qtdeDados;
    int VEZES = argp->repeticoes;
    short int tabela[tamanho];

  do
  {
   // preenche(tabela);
    preenche(tabela, tamanho);

    inicio = clock();
    quickSort(tabela, tamanho);
    fim = clock();

    diferenca = (double) (fim - inicio) / CLOCKS_PER_SEC;
    soma = soma + diferenca;
```
Arquivo "qsrpc_servidor.c" (parte 2/3), ao ser compilado com o "qsrpc_svc.c" e "qsrpc_xdr.c" gera o aplicativo equivalente ao "quickSortRPC_SV".

```
    if (primeiraVez)
    {
      maior = menor = diferenca;
      primeiraVez = 0;
    }
    else
    {
      if(diferenca > maior)
      {
        maior = diferenca;
      }
      if(diferenca < menor)
      {
        menor = diferenca;
      }

    }

    printf("\nTempo gasto para %i dados: %5.2lf segundos", tamanho, diferenca);
    vezes++;
  }
  while(vezes < VEZES);

   printf("\nMaior tempo gasto para %i dados: %5.2lf segundos", tamanho, maior);
   printf("\nMenor tempo gasto para %i dados: %5.2lf segundos", tamanho, menor);
   printf("\nTempo medio gasto para %i dados: %5.2lf segundos", tamanho, soma/(double)VEZES);
   progfim = clock();
   retorno = (double) (progfim - progini)/ CLOCKS_PER_SEC;
   printf("\n\nTérmino contagem programa quicksort. Tempo %5.2lf segundos",(double) (progfim - progini)
            / CLOCKS_PER_SEC);
    printf("\n");
  return(&retorno);


}
```

Arquivo "qsrpc_servidor.c" (parte 3/3), ao ser compilado com o "qsrpc_svc.c" e "qsrpc_xdr.c" gera o aplicativo equivalente ao "quickSortRPC_SV".

- Código fonte do protocolo.

```
/* Estrutura que ira passar os parametros */
struct params {
    int qtdeDados;
    int repeticoes;
};

/* Definicao do programa */
program QUICKSORT_RPC { /*nome do programa*/
    version QSRPC_V1 { /*versao do programa*/
        double QUICKSORT(params) = 1; /* nome procedimento*/
    } = 1;
} = 0x20000001;
```

Arquivo "qsrpc.x", é processado pelo RPCGEN gerando os arquivos "qsrpc_clnt.c", "qsrpc_svc.c", "qsrpc_xdr.c" e "qsrpc.h"

- Interface para interoperabilidade XDR gerada pelo RPCGEN

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "qsrpc.h"

bool_t
xdr_params (XDR *xdrs, params *objp)
{
    register int32_t *buf;

     if (!xdr_int (xdrs, &objp->qtdeDados))
         return FALSE;
     if (!xdr_int (xdrs, &objp->repeticoes))
         return FALSE;
    return TRUE;
}
```

Arquivo "qsrpc_xdr.c" gerado automaticamente pelo RPCGEN.

- Código fonte da biblioteca.

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#ifndef _QSRPC_H_RPCGEN
#define _QSRPC_H_RPCGEN

#include <rpc/rpc.h>


#ifdef __cplusplus
extern "C" {
#endif


struct params {
    int qtdeDados;
    int repeticoes;
};
typedef struct params params;

#define QUICKSORT_RPC 0x20000001
#define QSRPC_V1 1

#if defined(__STDC__) || defined(__cplusplus)
#define QUICKSORT 1
extern  float * quicksort_1(params *, CLIENT *);
extern  float * quicksort_1_svc(params *, struct svc_req *);
extern int quicksort_rpc_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define QUICKSORT 1
extern  float * quicksort_1();
extern  float * quicksort_1_svc();
extern int quicksort_rpc_1_freeresult ();
#endif /* K&R C */

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_params (XDR *, params*);

#else /* K&R C */
extern bool_t xdr_params ();

#endif /* K&R C */
```

Arquivo "qsrpc.h" (parte 1/2) gerado automaticamente pelo RPCGEN.

```
#ifdef __cplusplus
}
#endif

#endif /* !_QSRPC_H_RPCGEN */
```

Arquivo "qsrpc.h" (parte 1/2) gerado automaticamente pelo RPCGEN

- Código fonte do apêndice do cliente.

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <memory.h> /* for memset */
#include "qsrpc.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

float *
quicksort_1(params *argp, CLIENT *clnt)
{
    static float clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, QUICKSORT,
        (xdrproc_t) xdr_params, (caddr_t) argp,
        (xdrproc_t) xdr_float, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}
```

Arquivo "qsrpc_clnt.c" gerado automaticamente pelo RPCGEN.

- Código fonte do apêndice do servidor.

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "qsrpc.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void
quicksort_rpc_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        params quicksort_1_arg;
    } argument;
    char *result;
    xdrproc_t _xdr_argument, _xdr_result;
    char *(*local)(char *, struct svc_req *);

    switch (rqstp->rq_proc) {
    case NULLPROC:
        (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
        return;

    case QUICKSORT:
        _xdr_argument = (xdrproc_t) xdr_params;
        _xdr_result = (xdrproc_t) xdr_float;
        local = (char *(*)(char *, struct svc_req *)) quicksort_1_svc;
        break;

    default:
        svcerr_noproc (transp);
        return;
    }
    memset ((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
        svcerr_decode (transp);
        return;
    }
```

Arquivo "qsrpc_svc.c" (parte 1/2) gerado automaticamente pelo RPCGEN.

```
    result = (*local)((char *)&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result, result)) {
        svcerr_systemerr (transp);
    }
    if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
        fprintf (stderr, "%s", "unable to free arguments");
        exit (1);
    }
    return;
}

int
main (int argc, char **argv)
{
    register SVCXPRT *transp;

    pmap_unset (QUICKSORT_RPC, QSRPC_V1);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create udp service.");
        exit(1);
    }
    if (!svc_register(transp, QUICKSORT_RPC, QSRPC_V1, quicksort_rpc_1, IPPROTO_UDP)) {
        fprintf (stderr, "%s", "unable to register (QUICKSORT_RPC, QSRPC_V1, udp).");
        exit(1);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        fprintf (stderr, "%s", "cannot create tcp service.");
        exit(1);
    }
    if (!svc_register(transp, QUICKSORT_RPC, QSRPC_V1, quicksort_rpc_1, IPPROTO_TCP)) {
        fprintf (stderr, "%s", "unable to register (QUICKSORT_RPC, QSRPC_V1, tcp).");
        exit(1);
    }

    svc_run ();
    fprintf (stderr, "%s", "svc_run returned");
    exit (1);
    /* NOTREACHED */
}
```

Arquivo "qsrpc_svc.c" (parte 2/2) criado automaticamente pelo RPCGEN.

- Código fonte do aplicativo "quickSortSeq"

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>



void qs(short int tabela[], int ini, int fim)
{
  int i, j;
  int pivot;
  short int y;

  i = ini;
  j = fim;
  pivot = tabela[(ini+fim)/2];
  do
  {

    while((tabela[i]<pivot) && (i<fim))
    {
      i++;
    }

    while((pivot<tabela[j]) && (j>ini))
    {
      j--;
    }
    if(i<=j)
    {
      y = tabela[i];
      tabela[i] = tabela[j];
      tabela[j] = y;
      i++;
      j--;
    }
  }
  while(i<=j);

  if(ini<j)
  {
    qs(tabela, ini, j);
  }
```

Arquivo "quickSortSequencial.c" (parte 1/3) , ao ser compilado gera o aplicativo equivalente ao "quickSortSeq"

```
  if(i<fim)
  {
    qs(tabela, i, fim);
  }
}

void quickSort(short int tabela[], int tamanho)
{
  qs(tabela, 0, tamanho-1);
}

// Preenche a tabela com dados aleatórios
void preenche(short int tabela[], int qtde)
{
  int i;
  for(i = 0; i < qtde; i++)
  {
    tabela[i] = rand()%10000;
  }
}


main(int argc, char * argv[])
{


  int vezes = 0, primeiraVez = 1;
  int i;
  clock_t inicio, fim, progini, progfim;
  double diferenca, maior, menor, media, soma = 0;

  /*Inicio contagem tempo total do aplicativo*/
  progini = clock();
  printf("\nInício contagem tempo total do aplicativo");

  if ((argc !=3) || (!isdigit(argv[1][0])) || (!isdigit(argv[2][0]))){
    fprintf(stderr, "Uso: %s qtdeDados qtdeRepeticoes\n", argv[0]);
    exit(1);
  }

  int tamanho = atoi(argv[1]);
  int VEZES = atoi(argv[2]);
  short int tabela[tamanho];
```

Arquivo "quickSortSequencial.c" (parte 2/3) , ao ser compilado gera o aplicativo equivalente ao "quickSortSeq"

```c
do
{
 // preenche(tabela);
  preenche(tabela, tamanho);

  inicio = clock();
  quickSort(tabela, tamanho);
  fim = clock();

  diferenca = (double) (fim - inicio) / CLOCKS_PER_SEC;
  soma = soma + diferenca;
  if (primeiraVez)
  {
    maior = menor = diferenca;
    primeiraVez = 0;
  }
  else
  {
    if(diferenca > maior)
    {
      maior = diferenca;
    }
    if(diferenca < menor)
    {
      menor = diferenca;
    }

  }

  printf("\nTempo gasto para %i dados: %5.2lf segundos", tamanho, diferenca);
  vezes++;
}
while(vezes < VEZES);


printf("\nMaior tempo gasto para %i dados: %5.2lf segundos", tamanho, maior);
printf("\nMenor tempo gasto para %i dados: %5.2lf segundos", tamanho, menor);
printf("\nTempo medio gasto para %i dados: %5.2lf segundos", tamanho, soma/(double)VEZES);
progfim = clock();
printf("\n\nTérmino contagem programa. Tempo total %5.2lf segundos",(double) (progfim  - progini)
  / CLOCKS_PER_SEC);

printf("\n");

}
```

Arquivo quickSortSequencial.c" (parte 3/3) , ao ser compilado gera o aplicativo equivalente ao "quickSortSeq"

APÊNDICE B - Amostras do estudo de caso.

- "quickSortSeq" em execução

```
[root@localhost quicksortSeq]# ./quickSorteSeq 5000000 20

Início contagem tempo total do aplicativo
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.48 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
_
```

Início execução do aplicativo "quickSortSeq"

```
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.48 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Maior tempo gasto para 5000000 dados:  0.48 segundos
Menor tempo gasto para 5000000 dados:  0.46 segundos
Tempo medio gasto para 5000000 dados:  0.46 segundos

Término contagem programa. Tempo total 10.56 segundos
[root@localhost quicksortSeq]# _
```

Final execução do aplicativo "quickSortSeq"

- "quickSortRPC" junto ao "quickSortRPC_SV" com execução em uma mesma máquina.

```
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Maior tempo gasto para 5000000 dados:  0.47 segundos
Menor tempo gasto para 5000000 dados:  0.45 segundos
Tempo medio gasto para 5000000 dados:  0.46 segundos

Término contagem programa quicksort. Tempo 10.43 segundos


Término contagem programa quicksort. Tempo total 10.43 segundos
Chamada finalizada


Término contagem programa RPC. Tempo 10.46 segundos
[root@localhost quicksortRPC]# ./quicksortRPC localhost 5000000 20_
```

Final execução do conjunto "quickSortRPC" e "quickSortRPC_SV" na mesma máquina (cliente e servidor).

- "quickSortRPC" com execução em máquinas separadas.

```
[root@localhost quicksortRPC]# ./quicksortRPC 192.168.1.102 5000000 20
Chamada requisitada


Término contagem programa quicksort. Tempo total 10.39 segundos
Chamada finalizada


Término contagem programa RPC. Tempo 10.41 segundos
[root@localhost quicksortRPC]# _
```

Início e fim de execução "quickSortRPC" em máquina cliente.

- "quickSortRPC_SV" com execução em máquinas separadas.

```
[root@localhost quicksortRPC]# ./quicksortRPC_SV &
[2] 2218
[root@localhost quicksortRPC]#
Início contagem tempo total do aplicativo quicksort
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.48 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
_
```

Início do atendimento de uma chamada remota.

```
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.47 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.44 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Tempo gasto para 5000000 dados:  0.45 segundos
Tempo gasto para 5000000 dados:  0.46 segundos
Maior tempo gasto para 5000000 dados:  0.47 segundos
Menor tempo gasto para 5000000 dados:  0.44 segundos
Tempo medio gasto para 5000000 dados:  0.46 segundos

Término contagem programa quicksort. Tempo 10.45 segundos
_
```

Final do atendimento de uma chamada remota.