Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

"Uma Arquitetura de Proxy com Prioridades de Serviços para Chamadas Remotas de Procedimentos de Tempo Real"

Pedro Northon Nobile

São Carlos
Junho/2007

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

N745ap

Nobile, Pedro Northon.
Uma arquitetura de proxy com prioridades de serviços para chamadas remotas de procedimentos de tempo real / Pedro Northon Nobile. -- São Carlos : UFSCar, 2008.
127 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2007.

1. Sistemas operacionais distribuídos (Computadores). 2. Tempo real. 3. Qualidade de serviço. 4. Escalonamento. 5. Teoria das filas. I. Título.

CDD: 005.44 (20ª)

Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

"Uma Arquitetura de Proxy com Prioridades de Serviços para Chamadas Remotas de Procedimentos de Tempo Real"

PEDRO NORTHON NOBILE

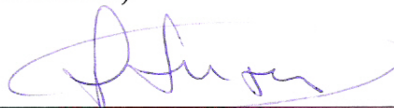
Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Membros da Banca:

Prof. Dr. Luis Carlos Trevelin
(Orientador – DC/UFSCar)

Prof. Dr. Célio Estevan Moron
(Co-Orientador – DC/UFSCar)

Prof. Dr. Hélio Crestana Guardia
(DC/UFSCar)



Prof. Dr. Hermes Senger
(UNISANTOS)

São Carlos
Junho/2007

A minha família, em especial meu pai, Carlos Roberto Nobile, que muita falta me faz neste momento feliz.

Agradecimentos

Agradeço, sinceramente e em primeiro lugar, a Deus, por me dar ciência, saúde e persistência para perseguir meus objetivos.

Aos meus pais, Luiza e Carlos, que acreditaram em mim desde o início e não mediram esforços para tornar meus sonhos possíveis.

Ao meu irmão Fabio que esteve sempre do meu lado e apoiou-me nos momentos mais difíceis da minha trajetória. Jamais vou esquecer os momentos que só pude contar com ele.

A toda minha família pelo apoio e reconhecimento de meus esforços.

A minha noiva e futura esposa Marcela pela paciência, pelos carinhos e pelo meu maravilhoso filho João Pedro, que sempre renovou minhas esperanças.

Aos professores Luís Carlos Trevelin e Célio Estevan Moron pela orientação e suporte ao meu trabalho, além da compreensão e confiança em mim depositados.

A todos os colegas do Laboratório Danilo, Ricardo, Vinícius, Marsola e todos os outros pelo apoio e discussões sem fundamento que alegravam meus dias.

A todos os meus amigos de São Carlos e Rancharia que foram importantes na minha vida durante o meu trabalho. Em especial, agradeço aos meus amigos Andreza e Rigolin que, sem dúvida, contribuíram com sua amizade nos momentos mais difícies que passei. Momentos que jamais serão apagados.

Aos professores do grupo de Sistemas Distribuídos e Redes, que contribuíram com sugestões e críticas. Ao Departamento de Computação - DC, formado por todos os funcionários, pela colaboração e agilidade sempre que precisei de seus serviços.

A CNPq pelo suporte financeiro.

E a todos que participaram direta ou indiretamente deste trabalho.

Sumário

Lista de Figuras **viii**

Lista de Tabelas **ix**

Lista de Siglas **xi**

Resumo **xv**

Abstract **xvii**

1 Introdução **1**
1.1 Projeto Metodológico 7

2 Fundamentos de Qualidade de Serviço **11**
2.1 Conceito de Qualidade de Serviço 12
2.2 Garantia de Qualidade de Serviço 13
2.3 Serviços Integrados 14
2.3.1 Resource Reservation Protocol 15
2.3.1.1 Visão Geral da Operação do RSVP 16
2.4 Serviços Diferenciados 18
2.4.1 DiffServ 19

3 Fundamentos de Computação Distribuída **25**
3.1 Remote Procedure Call (RPC) 26
3.1.1 Aspectos de Projeto de RPC 26
3.1.2 Operação em RPC 27
3.2 Distributed Component Object Model (DCOM) 29
3.2.1 Trasparência de Localização 32
3.2.2 Arquitetura DCOM 33
3.3 Commom Object Request Broker Architecture (CORBA) 34
3.3.1 Definição de Interfaces 37
3.3.2 Implementação de Clientes e Objetos 38
3.4 Remote Method Invocation (RMI) 40
3.5 Gerador de *Graphical User Interface (GUI)* com suporte a *Real Time (RT)-RPC* 42

4 Fundamentação Matemática **49**
4.1 Teoria de Filas 49
4.1.1 Filas 50
4.1.2 Tipos de Filas 53
4.1.3 Medidas de Interesse 55
4.2 Séries Temporais 65
4.2.1 Método Box-Jenkins 66
4.2.1.1 Modelos Estacionários 66
4.2.1.2 Modelo Não Estacionário 70
4.2.1.3 Modelo Sazonal 72
4.2.2 Etapas de modelagem Box-Jenkins 73

5 Arquitetura de Proxy com Prioridades de Serviços para Chamadas Remotas de Procedimento de Tempo Real **75**
5.1 Queuing System Manager 77
5.1.1 Client Listener 78
5.1.2 Server Listener 79
5.2 Queueing System Processor 80
5.3 Data Manager 82
5.4 Decision Manager 82
5.4.1 Prediction System 83
5.4.1.1 Ajuste do Prediction System 84
5.4.2 Decision System 87
5.4.3 Políticas e Policy Database 88

6 Estudo de Caso **95**
6.1 Desenvolvimento da Aplicação de Teste 96
6.1.1 Módulo Cliente da Aplicação de Teste 97
6.1.2 Módulo Servidor da Aplicação de Teste 98
6.2 Análise Preliminar do Comportamento do Ambiente de Rede 99
6.3 Tempo de Processamento 99
6.3.1 Medidas de Tempo de Processamento 101
6.4 Modelos de Previsão 103
6.4.1 Modelo de Previsão Para Mensagens de Média Alta e Alta Prioridade 104
6.4.2 Modelo de Previsão Para Mensagens de Baixa e Média Baixa Prioridade 108
6.5 Comparação entre Alocação Estática e Dinâmica de Processadores 112
6.6 Comparação entre Políticas de Inserção FIFO, por Deadline e por Prioridade-Deadline 116

7 Conclusão **119**
7.1 Expectativa de Uso da Arquitetura 120
7.2 Trabalhos Futuros 121

Lista de Figuras

1.1 Um exemplo de Home Network. 2
1.2 Ambientes de Rede de RT-RPCs. 6

2.1 Host e Roteador com Suporte a RSVP. 16
2.2 Operação RSVP. 18
2.3 Cabeçalho IPv4. 20
2.4 Esquema de um Domínio DiffServ. 20
2.5 Roteador com Suporte a DiffServ. 22

3.1 Estrutura de uma RPC. 29
3.2 Comunicação de Componentes Entre Processos de Uma Mesma Máquina. . 30
3.3 Estrutura de GUID. 31
3.4 Interação Cliente-Componente Utilizando COM. 32
3.5 Interação Cliente-Componente Utilizando DCOM. 34
3.6 Arquitetura do Modelo de Referência OMA. 35
3.7 Arquitetura CORBA. 36
3.8 Repositório de Interfaces e Implementação. 38
3.9 Implementação Típica de Cliente CORBA. 39
3.10 Implementação Típica de Objeto CORBA. 40
3.11 Exemplo de Aplicação RMI. 42
3.12 Códigos Gerados pelo Gerador de GUI com Suporte a RT-RPC. 44
3.13 Estrutura de Mensagem RPC. 45
3.14 Estrutura de Mensagem Promise. 47
3.15 Estrutura de Mensagens de Confirmação e Cancelamento. 47

4.1 Modelo de Fila. 50
4.2 Sistema Único Estágio. 53
4.3 Sistema Múltiplo Estágio. 54
4.4 Sistema Único Estágio Paralelo. 54
4.5 Sistema Multicanal Único Estágio. 55
4.6 Sistema Multifilas. 55
4.7 Sistema Discriminatório de Clientes. 56
4.8 Eventos Sistema de Filas. 56
4.9 Distribuição Acumulada de Clientes no Sistema. 59

5.1 Arquitetura do Proxy com QoS para RT-RPCs. 76

5.2 Módulos do Decision Manager. 83
5.3 Arquitetura do Proxy Coletor. 85
5.4 Trecho da Policy Database. 89

6.1 Comunicação entre aplicação cliente e servidor. 97
6.2 Distribuição dos Tempos de Processamento. 102
6.3 Quantidade de RT-RPCs de Alta e Média Alta Prioridade Observada. . . . 105
6.4 Função de Auto-Correlação para RT-RPCs de Alta Prioridade. 105
6.5 Função de Auto-Correlação Parcial para RT-RPCs de Alta Prioridade. . . 106
6.6 Função de Auto-Correlação para os Resíduos do Modelo Estimado AR(4). 107
6.7 Comparação entre Valores Observados e Valores Previstos pelo Modelo AR(4). 108
6.8 Quantidade de RT-RPCs de Média Baixa e Baixa Prioridade Observada. . 109
6.9 Função de Auto-Correlação para RT-RPCs de Média Baixa e Baixa Prioridade. 109
6.10 Função de Auto-Correlação Parcial para RT-RPCs de Média Baixa e Baixa Prioridade. 110
6.11 Função de Auto-Correlação para os Resíduos do Modelo Estimado AR(3). 111
6.12 Comparação entre Valores Observados e Valores Previstos pelo Modelo AR(3). 111
6.13 Comparação entre Alocações Estáticas de 1 e 5 Processadores. 113
6.14 Comparação entre Alocação Estática de 5 processadores e Alocação Dinâmica. 114
6.15 Comparação entre Políticas de Inserção. 117

Lista de Tabelas

2.1 Classes de serviço para comportamento de encaminhamento garantido . . . 22

3.1 Interação Típica Entre Cliente e Servidor COM. 31
3.2 Tipos Básicos de Java. 44
3.3 Descrição das Partes da Estrutura de Mensagem RPC. 46
3.4 Descrição das Partes da Estrutura de Mensagem de Confirmação ou Cancelamento. 48

4.1 Tabela de Dados do Exemplo. 62
4.2 Tabela de Desempenho da Primeira Questão. 63
4.3 Tabela de Desempenho da Segunda Questão. 63
4.4 Tabela de Desempenho da Terceira Questão. 64
4.5 Padrões de Auto-Correlação e Auto-Correlação Parcial. 74

Lista de Siglas

ACF Função de Autocorrelação

AF Encaminhamento Garantido

AIC Critério de Informação de Akaike

API Interface de Programação de Aplicações

AR Auto-regressivo

ARIMA Auto-Regressivo Integrado Média Móvel

ARMA Auto-Regressivo Média Móvel

ATM Asynchronous Transfer Mode

AWT Abstract Window Tool

BA Behaviour Agregate

CL Ouvidor Cliente

COM Component Object Model

CORBA Commom Object Request Broker Architecture

DB Base de Dados

DCOM Distributed Component Object Model

DeM Gerenciador de Decisão

DiffServ Serviços Diferenciados

DII Dynamic Invocation Interface

DLL Dynamic Link Library

DM Gerenciador de Dados

DS Sistema de Decisão

DSCP Differentiated Services Code Point

DSI Dynamic Skeleton Interface

EF Expedited Forwarding

FCFS First Come, First Served

FCLS First Come, Last Served

FIFO First In, First Out

FILO First In, Last Out

GUI Graphical User Interface

GUID Globally Unique Identifier

HAN Home Area Network

IDE Interface Development Environment

IDL Interface Definition Language

IntServ Integrated Services

IP Internet Protocol

ISP Provedor de Serviços de Internet

JDK Kit Java de Desenvolvimento

JVM Máquina Virtual Java

LPC Chamada de Procedimento Local

MA Média Móvel

MF MultiField classifier

OMA Object Management Architecture

OMG Object Management Group

ORB Object Request Broker

PACF Função de Auto-correlação Parcial

PDB Base de Dados de Políticas

PHB Comportamento por Host

PS Sistema de Previsão

QoS Qualidade de Serviço

QSM Queueing System Manager

QSP Queueing System Processor

RMI Remote Method Invocation

RPC Remote Procedure Call

RSVP Resource Reservation Protocol

RT Real Time

SARIMA Seasonal Auto-Regressive Integrated Moving Average

SEPT Shortest Expected Processing Time

SI Sistemas Integrados

SIRO Service In Random Order

SL Server Listener

SLA Contrato de Nível de Serviço

SPT Menor Tempo de Processamento

SSD Sistemas de Suporte à Decisão

TCP Transmission Control Protocol

TDM Multiplexação por Divisão de Tempo

TDMA Acesso Múltiplo por Divisão de Tempo

TOS Tipo de Serviço

Resumo

Nos últimos anos tem-se observado um crescimento acentuado de aplicações de tempo real, principalmente de aplicações que envolvem multimídia (e.g. streamming de vídeo e voz). Fatores como a popularização de redes de banda larga (e.g. xDSL) e das redes sem fio(e.g. família 802.11 e 802.15), juntamente com o surgimento de melhores dispositivos e a necessidade de comunicação cada vez maior por parte dos usuários, têm contribuído para tal crescimento. Atualmente, com o surgimento de novos conceitos de comunicação como as home networks, é possível prever que em um futuro próximo, a comunicação entre aplicações de tempo real fará parte do cotidiano dos usuários. Antecipando uma solução para essa previsão, este trabalho propõe uma arquitetura de proxy com prioridades de serviços para chamadas remotas de procedimento de tempo real, que utiliza os volumes anteriores de tráfego para fazer previsões sobre o estado futuro de ocupação da rede, selecionando as políticas de atuação mais adequadas para que o proxy se adapte aos futuros estados do ambiente.

Abstract

During the last few years is possible to observ an significant increase of real time applications, , mainly of multimedia applications (e.g. video and voice streamming). Factors such as the popularization of broadband (e.g. XDSL) and wireless (e.g. 802.11 familiy and 802.15 pattern) networks, associated to the appearance of better devices and the increase of user's communications need, have contributed. Currently, with the appearance of new concepts of communications like home networks, it is possible to predict that real time applications will be part of user's daily. Anticipating a solution to that predict, this work proposes a proxy architecture with priorities of services for real time remote procedure calls, that uses the earlier amount of traffic to predict the future occupation network, and choose the most appropriate policy to adapt the proxy to the environment future states.

Capítulo 1

Introdução

Nos últimos anos tem-se observado um crescimento acentuado de aplicações de tempo real, principalmente de aplicações que envolvem multimídia como, por exemplo, aplicações de *streamming* de vídeo e voz (*e.g.* vídeo sob demanda e voz sobre *IP*), que se tornam cada vez mais populares para compartilhamento de conteúdos, em especial, de entretenimento.

Diversos são os fatores que têm contribuído para o crescimento dessas aplicações de tempo real. Um dos fatores que tem contribuído para este crescimento é a popularização de redes de banda larga (*e.g.* xDSL) e das redes sem fio, como a família de padrões 802.11 (Crow et al., 1997) e o padrão 802.15(Blu, 2004). Enquanto as redes de banda larga melhoram qualitativamente o acesso a conteúdos remotos, as redes sem fio conseguem conciliar a mobilidade e a conectividade, onde um usuário pode locomover-se dentro de uma área permanecendo conectado a serviços remotos.

Outro fator que deve ser ressaltado é o surgimento de melhores dispositivos, com maior capacidade de processamento e memória, que permitem a implementação de aplicações mais complexas e mais exigentes quanto a desempenho. Assim como os outros fatores citados, a necessidade intrínseca que a maioria dos usuários tem de acesso à informação é um dos contribuintes mais naturais para o crescimento desse tipo de aplicações.

Nesse contexto, as características das aplicações de tempo real quanto a distribuição das informações também têm sofrido mudanças nos últimos tempos. As aplicações que há algum tempo comportavam-se como aplicações *unicast* (*i.e.* um-para-um), atualmente, comportam-se como *multicast* (*e.g.* vídeo-conferência e jogos *on-line*), onde os dados enviados por um emissor podem ter como destino vários receptores e vice-versa.

Juntamente com o crescimento das aplicações multimídia de tempo real, é possível observar a emergência de um novo conceito de rede, as *Home Area Network (HAN)*. As *HANs* constituem redes domésticas que envolvem os aparelhos eletrônicos de uma residência (*i.e.* televisores, computadores e telefones) interligados por meio das redes elétrica, de telefonia, e de dados, podendo esta última ser cabeada ou não. Nas *HANs* os dispositivos domésticos formam uma rede bastante heterogênea conectada a um único *gateway* residencial, como pode ser observado na Figura 1.1.

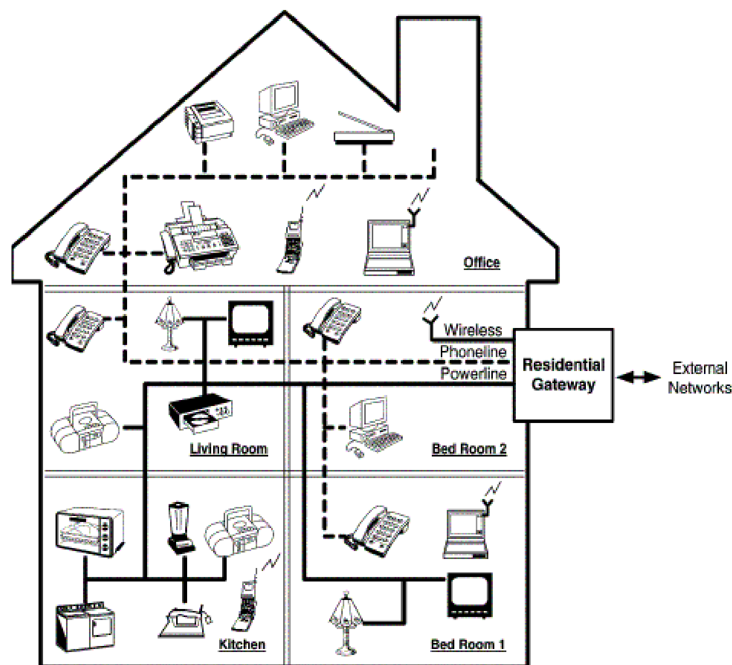


Figura 1.1: Um exemplo de Home Network.

Fonte: Ganz et al. (2003)

O surgimento das *home networks* reforça o conceito de computação pervasiva (Satyanarayanan, 2001), onde os usuários passam a ser envolvidos por dispositivos computacionalmente ativos conectados permanentemente a alguma rede de comunicação.

Vários trabalhos estão sendo desenvolvidos com relação a esse novo conceito, como, por exemplo, em Amigo (2004) que implementa um *middleware* e serviços de usuário para criar um ambiente de rede doméstica inteligente. Já em (Ganz et al., 2003) é proposto um *framework* para suportar qualidade de serviço em redes domésticas, enquanto Kim et al. (2002), em seu trabalho, propõem e implementam um *middleware* para rede doméstica usando *Universal Plug-and-Play*. No entanto, nesses trabalhos, não é proposta nenhuma forma eficiente de controle e/ou diferenciação do tráfego de tempo real que percorre a rede.

Além das *home networks*, espera-se que outros tipos de rede como redes laboratoriais e redes corporativas façam uso cada vez maior de aplicações de tempo real. O aumento na utilização de aplicações com requisitos de tempo real esbarra na necessidade de garantir *Qualidade de Serviço (QoS)* para essas aplicações. Atualmente muitos trabalhos concentram seus esforços na garantia de *QoS* para aplicações de tempo real, sobretudo, para aplicações multimídia interativas. Em geral, aplicações de tempo real são sensíveis e pouco tolerantes aos atrasos e às eventuais perdas de pacotes que podem ocorrer durante a transmissão, exigindo uma qualidade mínima de serviço para sua execução efetiva.

No atual estado da arte, a maioria das soluções propostas para resolver o problema de garantia de *QoS* das aplicações multimídia são baseadas em mecanismos *Serviços Diferenciados (DiffServ)* (Aimoto e Miyake, 2000) (Black et al., 1998) ou *Integrated Services (IntServ)* (Zhang et al., 2002) (Braden, 1997). Muitos trabalhos apresentam soluções destinadas a alguns tipos específicos de tráfego, mas poucos consideram o suporte a comunicação por *RPCs*, principalmente *RT-RPCs*. Geralmente, esses trabalhos propõem soluções na forma de algoritmos de controle, *frameworks* de *QoS*, protocolos de serviço e arquiteturas de sistemas.

As *RT-RPCs* são bastante utilizadas na implementação de aplicações distribuídas e possuem características diferentes quando comparadas às aplicações sem requisitos de tempo real. Para as aplicações multimídia, mesmo as interativas, pequenos atrasos ou perdas que ocorrem durante a transmissão não alteram fundamentalmente o sucesso da execução da aplicação. Por outro lado, uma *RT-RPC*, cuja exatidão depende tanto da execução correta, quanto do tempo em que o resultado é produzido, requer tempos resposta mais rigorosos, e, dessa forma, é muito menos tolerantes à ocorrência de falhas.

Em seu trabalho Lee (1996) propõe um sistema de comunicação sobre redes *Acesso Múltiplo por Divisão de Tempo (TDMA)* capaz de suportar o tráfego de *RPCs* de tempo real. Nesse sistema, além das mensagens de requisição e resposta, é adicionada uma nova mensagem (*ACK*) com função de notificação. Essa mensagem é enviada pelo servidor para notificar o cliente se uma requisição *RPC* pode ou não ser atendida antes do vencimento do *deadline*. Apesar da mensagem *ACK* evitar o desperdício de recursos para armazenamento de uma requisição que não pode ser atendida, a adição de mais uma mensagem acrescenta uma sobrecarga à comunicação. Além disso, o sistema reserva tempo do *deadline* para a notificação, restringindo ainda mais o tempo de execução. O autor propõe uma implementação simples no servidor, uma vez que organiza as *RPCs* sem considerar suas características individuais. Esse servidor também não possui mecanismo de previsão de chegadas de requisições, e por isso não antecipa a alocação de recursos para atender a demanda de requisições futura.

Wang et al. (2005) propõem uma arquitetura de *proxy* robusta e *environment-aware* que fornece qualidade de serviço para aplicações legadas sem a necessidade de adição de parâmetros de *QoS*. Com isso, a arquitetura permite a coexistência de aplicações legadas com aquelas que já possuem tais parâmetros. O autor fornece uma aplicação gráfica onde usuários de aplicações antigas podem especificar os requisitos de *QoS* de cada aplicação e propagá-los para o *proxy* por meio de perfis de usuário. Assim como os trabalhos

anteriores, essa arquitetura também não considera o tráfego de *RPCs*, abordando apenas o fluxo de aplicações convencionais (i.e. *HTTP*, *FTP* e multimídia).

Kummel et al. (1997) propõem uma abordagem de *QoS* para comunicação de *RPCs*, em redes *Asynchronous Transfer Mode (ATM)*, com garantia total de banda para os *hosts* envolvidos. Esse trabalho foca especificamente na comunicação constante entre dois *hosts*. A abordagem proposta usa da manutenção de comunicação entre dois *hosts* para diminuir a quantidade de conexões lógicas a serem estabelecidas e utilizar o tempo que seria desprendido, com a configuração da conexão, para a transmissão de *RPCs*. Contudo, além de não abordar *RPCs* com características de tempo real, a *QoS* é fornecida de forma igualitária para todas as mensagens *RPCs* de um fluxo, desprezando características individuais de cada chamada.

Alguns autores como Lei et al. (2002) e Hwang e Tseng (2005) propõem soluções para garantia de *QoS* na forma de arquiteturas de *gateway*. Esses autores apresentam arquiteturas que utilizam classificação de tráfego e gerenciamento racional da largura de banda disponível. Nesses trabalhos, a classificação do tráfego é definida em algumas classes de *QoS* separadas pelo tipo da aplicação. No entanto, nenhuma dessas classes engloba aplicações *RPCs*, classificadas como tráfego de baixa prioridade ou simplesmente como *best-effort*.

Num ambiente onde muitos dispositivos compartilham uma rede de acesso e o mesmo *gateway*, e realizam *RT-RPCs* destinadas a servidores de outra rede, o congestionamento causado pelo volume de chamadas que passa pelo *gateway* constitui um gargalo no cumprimento dos *deadlines*, principalmente para aquelas requisições de maior prioridade e requisitos de tempos mais críticos.

Neste contexto, é necessário que o *gateway* faça distinção entre as diferentes características de cada *RT-RPC*, e com base nessas informações privilegie aquelas de características mais rígidas. Todavia, está não é uma tarefa simples, uma vez que, possivelmente, o *gateway* possui recursos limitados e quando sujeito a um grande volume de chamadas, de-

verá alocar os recursos necessários para atender a demanda, além de organizar e escolher quais chamadas devem ser transmitidas. A Figura 1.2 ilustra um ambiente exemplo onde dispositivos de uma rede realizam *RT-RPCs* para controle remoto de experimentos.

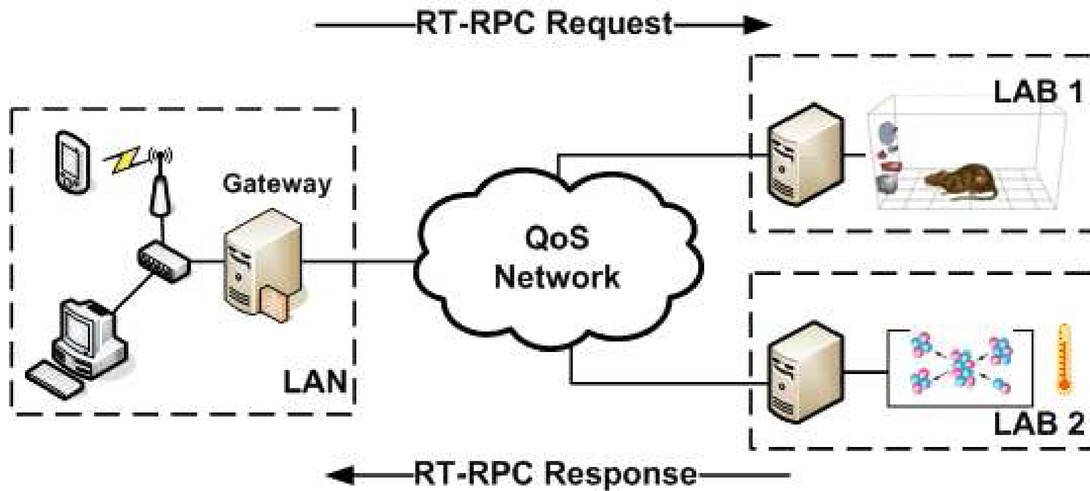


Figura 1.2: Ambientes de Rede de RT-RPCs.

Este trabalho concentra-se na comunicação de aplicações localizadas em redes fisicamente separadas por meio de *RT-RPCs*, e propõe uma arquitetura de *QoS proxy* transparente para *RT-RPCs* que utiliza os modelos de séries temporais *Box-Jenkins* (Box e Jenkins, 1976) para prever o volume e as características futuras do tráfego de *RT-RPCs* que, provavelmente, passarão pelo *proxy*. A previsão de tráfego permite a alocação antecipada e racional dos recursos necessários para atender a demanda prevista e a escolha de políticas que visam a adaptação do *proxy* aos estados do seu ambiente de rede. Para verificar a funcionalidade da arquitetura foi implementado um protótipo, o qual foi submetido a um ambiente de rede real que permitiu verificar seu comportamento sob diferentes volumes de *RT-RPCs*.

O texto está organizado da seguinte forma:

- No próxima seção do capítulo 1 (um) é apresentado o projeto metodológico que dirigiu o desenvolvimento do trabalho.

- No capítulo 2 (dois) são apresentados fundamentos de qualidade de serviço, onde são apresentados os principais padrões desenvolvidos nessa área.
- No capítulo 3 (três) são apresentados os fundamentos de computação distribuída, descrevendo os principais padrões de comunicação em ambientes distribuídos, além de descrever a ferramenta desenvolvida no trabalho Villela (2001).
- O capítulo 4 (quatro) apresenta a fundamentação matemática, onde discutem-se os conceitos de séries temporais e teoria de filas utilizados neste trabalho.
- No capítulo 5 (cinco) é apresentada a arquitetura proposta, bem como seus módulos e suas funcionalidades.
- No capítulo 6 (seis) é desenvolvido um estudo de caso utilizando um protótipo da arquitetura proposta, com resultados comparativos da utilização da mesma.
- Por fim, no capítulo 7 (sete) é apresentada a conclusão sobre o trabalho desenvolvido. Nesse mesmo capítulo ainda constam a expectativa de uso da arquitetura e os trabalhos futuros para complementação e melhoria da mesma.

1.1 Projeto Metodológico

Para este projeto foram definidos itens que buscam defini-lo de modo consistente, fornecendo delimitação e direcionamento da pesquisa. O itens definidos para este projeto são:

- Tema:
  - Acesso a serviços remotos de tempo real entre *hosts* localizados em redes fisicamente separadas com garantia de diferenciação de serviços.
- Título:
  - Uma Arquitetura de *Proxy* com Prioridades de Serviços para Chamadas Remotas de Procedimento de Tempo Real.

- Problema:
  - A ausência de mecanismos de *proxy* que garanta a qualidade de serviço com base na diferenciação de serviços, para o acesso a serviços remotos de tempo real por meio de chamadas remotas de procedimento, levando em conta as características de prioridade e *deadline* de cada chamada.
- Objetivo:
  - Criar uma arquitetura de *proxy* com prioridades de serviços, eficiente e escalável que permita acesso a serviços remotos de tempo real utilizando chamadas remotas de procedimento.
- Questão de pesquisa:
  - **É viável a criação e garantia de funcionalidade de tal arquitetura?**
- Hipóteses:
  - **H1**: É possível, por meio de um estudo prévio do ambiente de rede, criar modelos matemáticos de previsão que permitam antecipar o comportamento da rede num instante futuro, quanto ao fluxo e características (*i.e.* prioridade e *deadline*) de chamadas remotas de procedimento de tempo real.
  - **H2**: A utilização de filas,para organizar e classificar diferentes chamadas, permite que o fluxo de mensagens seja diferenciado e seja priorizado o tratatamento de mensagens com requisitos mais rígidos.
  - **H3**: É possível que por meio de previsões, sejam selecionadas políticas de comportamento internas à arquitetura para que haja uma transmissão satisfatória das chamadas remotas de procedimento de tempo real, além da adequação da arquitetura ao ambiente.
- Delimitação:

  - Este trabalho delimita-se a propor uma arquitetura de *proxy* que localiza-se nas bordas de uma rede da qual *hosts* podem executar ou realizar chamadas remotas de procedimento de tempo real. Supõe-se que o *proxy* deva estar conectado a uma rede com qualidade de serviço ou linha dedicada para garantir a transmissão entre o *host* que realiza a chamada e o *host* que executa a chamada.

- Relevância:

  - A criação desta arquitetura vai permitir que chamadas remotas de tempo real sejam divididas e encaminhadas de maneira inteligente, privilegiando chamadas de maior prioridade. Um *proxy* com tais características pode ser utilizado de diferenciação de serviços em uma rede com *gateway* compartilhado por vários usuários dispondo de serviços diversos com diferentes prioridades e requisitos temporais.

Capítulo 2

Fundamentos de Qualidade de Serviço

Este capítulo tem o propósito de introduzir fundamentos e dar um panorama do atual estado da arte com relação ao conceito de *QoS*.

A garantia de *QoS* é fundamental na execução de aplicações distribuídas de tempo real e tem ganhado cada vez mais destaque com o surgimento de aplicações multimídia de tempo real, as quais têm sido a mola propulsora para as pesquisas nessa área.

A convergência para o uso desses tipos de aplicações faz com que as redes sejam inundadas por diferentes tipos de tráfegos com diferentes requisitos. Essas aplicações possuem requisitos rígidos de tempo e são, em essência, pouco tolerantes a atrasos e perdas de dados que podem ocorrer durante a transmissão. Dessa forma, garantir a qualidade de serviço na transmissão é essencial para o sucesso ou não das mesmas.

A literatura traz diversos trabalhos na área de *QoS*, com propostas de novos padrões e arquiteturas que buscam garantir a qualidade de serviço para redes do tipo *best-effort* como, por exemplo, as redes *Internet Protocol (IP)* (Postel, 1981).

As próximas seções introduzem o conceito de *QoS* e mostram alguns importantes trabalhos que vem sendo desenvolvidos para essa área.

2.1 Conceito de Qualidade de Serviço

O termo *QoS* é um termo bastante amplo para diversas áreas de pesquisa, não existindo uma definição formal universalmente aceita. Diversas áreas da computação adotam este termo com significados, às vezes, não muito similares.

O conceito de *QoS* é mais amplamente estudado na área de redes de computadores e pode ser definido de diferentes maneiras. Tais definições quase sempre estão ligadas ao fornecimento de melhores serviços para transmissão sobre diferentes tecnologia de rede. Para *QoS*, os parâmetros considerados mais influentes são: largura de banda disponível, latência, variação de atrasos, perda de pacotes. Algumas aplicações são mais tolerantes com relação a variação desses parâmetros, enquanto outras são intolerantes a qualquer tipo de variação.

Segundo Melo (2001), Qualidade de Serviço é entendida como a capacidade da rede, através dos mecanismos de reserva de largura de banda e priorização de tráfego, em fornecer garantias de que determinados fluxos de tráfego irão ter tratamento diferenciado.

Em Ferguson e Huston (1999), qualidade de serviço é definida como a habilidade de um elemento da rede, seja uma aplicação, host, roteador, ou outro dispositivo, ter algum nível de garantia que seu tráfego e exigências de serviço podem ser satisfeitas.

QoS também pode ser definida como uma necessidade para permitir que todas as aplicações de comunicação entre computadores consigam definir suas necessidades e satisfazer seus requisitos (Nor, 1999).

Para Teitlebaum e Hans (1998), *QoS* é um termo que freqüentemente possui duplo significado: refere-se ao desempenho de uma rede em relação as necessidades de uma aplicação e ao conjunto de tecnologias que permitem a rede satisfazer estas garantias de desempenho.

A qualidade de serviço pode ser definida como a capacidade da rede prover serviço de encaminhamento de dados de forma consistente e previsível. Em outras palavras, é a capacidade de satisfação das necessidades das aplicações dos usuários (Sta, 1999).

Segundo Park (2005) *QoS* é definida sobre 2 (dois) pontos de visão: *QoS* do ponto de vista do usuário e do ponto de vista da rede. Do ponto de vista do usuário, *QoS* é a percepção dos usuários finais da qualidade que eles recebem do provedor para um serviço ou aplicação particular que eles assinam (*e.g.* voz, vídeo ou dados). Do ponto de vista da rede, o termo *QoS* refere-se a capacidade da rede prover a *QoS* observada pelos usuários finais.

Apesar das definições focarem em aspectos diferentes, em essência, elas são bastante convergentes. Neste trabalho a definição de *QoS* adotada é a de que: ***QoS* é a capacidade satisfazer a demanda de aplicações exigentes quanto a comunicação, sejam elas pouco ou não tolerantes a problemas de transmissão**.

2.2 Garantia de Qualidade de Serviço

A implementação de *QoS* torna-se necessária a medida que a largura de banda disponível não é suficiente para a demanda das aplicações. Uma das saídas para redes do tipo *best-effort* é o aumento da largura de banda a medida que a demanda das aplicações também aumenta. Esta alternativa é dispendiosa e pode tornar-se ineficiente com o tempo, uma vez que, com o crescente aumento de números de usuários e aplicações, e o caráter compartilhado dos recursos, o aumento da largura de banda teria que tender ao infinito.

Outro problema em relação ao aumento da largura de banda é a necessidade de propagar esse aumento para todas as redes pelas quais o fluxo de dados irá trafegar, na tentativa de evitar o provável congestionamento que pode ocorrer em qualquer ponto do caminho a ser percorrido.

Outra opção seria a alocação de canais dedicados, ou a utilização de alternativas que garantam a transmissão, para o usuário que utiliza aplicações que necessitem de qualidade de serviço. No entanto, esta opção, além de dispendiosa, vai de encontro com a impossibilidade de prever o aumento no número dessas aplicações e no desperdício da banda alocada para o canal quando este não estiver sendo utilizado.

Somente o aumento na largura de banda ou a contração de canais dedicados não é suficiente para garantir a qualidade de serviço. Outras alternativas devem ser consideradas levando-se em consideração a atual infra-estrutura disponível. Existem duas alternativas amplamente utilizadas para garantia de *QoS*:

- Serviços Integrados.
- Serviços Diferenciados.

Essas alternativas são contrárias ao conceito de *best-effort* e buscam a divisão racional dos recursos disponíveis para transmissão. As próximas seções detalham essas alternativas e trazem as principais soluções implementadas para cada uma delas.

2.3 Serviços Integrados

A abordagem baseada em Serviços Integrados (Melo, 2001) provê um conjunto de extensões para o modelo atualmente utilizado de *best-effort* utilizados nas redes *IP*. O *Sistemas Integrados (SI)* busca um tratamento diferenciado de diversos tráfegos por meio de reserva de recursos.

A reserva de recursos é o passo inicial executado pela aplicação antes de transmitir o fluxo. Os recursos devem ser previamente configurados ao longo do caminho do fluxo antes da transmissão dos dados. O modelo de *SI* otimiza a utilização da rede e dos recursos para aplicações pouco tolerantes a falhas e congestionamento. O bom funcionamento desse tipo de aplicação requer que os recursos estejam disponíveis e sejam suficientes quando forem necessários na transmissão.

Para conseguir cumprir com os requisitos das aplicações, o modelo de *SI* divide o tráfego de maneira racional para as aplicações tradicionais e as aplicações que necessitam de *QoS*. Para suportar o modelo de serviços integrados, um roteador precisa ser capaz de propiciar a *QoS* apropriada para cada fluxo de dados, de acordo com o modelo do serviço.

Neste modelo o recursos devem ser reservados fim-a-fim para garantir a *QoS* adequada. Dessa forma os estados de reserva devem ser mantidos pelo roteadores ao longo de todo caminho a ser percorrido pelo fluxo.

Para fazer reservas de recursos para um fluxo, a aplicação fonte deve prover uma especificação de fluxo. Uma especificação de fluxo consiste de uma caracterização de tráfego e requisitos de serviço para o fluxo. A caracterização do fluxo inclui a taxa máxima, a taxa média, o tamanho da rajada e os parâmetros de vazamento de balde; e os requisitos de serviço incluem a banda mínima necessária e os requisitos de desempenho (*e.g.* atraso, *jitter* e perda de pacotes). Os *SIs* utilizam o *Resource Reservation Protocol (RSVP)* para reserva de recursos para um fluxo (Park, 2005).

2.3.1 Resource Reservation Protocol

RSVP (RFC2205, 1997) é um protocolo de controle sobre *IP*. Este protocolo é utilizado para garantir *QoS* para aplicações multimídia ao longo do caminho de fluxo percorrido pelos pacotes entre a origem e o destino. O que este protocolo faz é criar um circuito com reserva de recursos necessários por onde os pacotes irão trafegar durante a transmissão.

Para que o *RSVP* consiga criar um circuito como descrito acima, todos os roteadores ao longo do caminho devem suportá-lo. O *RSVP* tem como único objetivo a reserva e controle de recursos ao longo do caminho, sendo que o mesmo utiliza outros protocolos para roteamento e transmissão. O controle envolve a manutenção dos recursos reservados e liberação quando os mesmos deixam de ser utilizados.

O *RSVP* é unidirecional, e sendo assim, reserva recursos apenas em uma das direções do fluxo. É orientado a receptor, uma vez que o mesmo deve iniciar a solicitação de

reserva. A solicitação é propagada exatamente pela rota inversa a do fluxo até chegar a fonte, ou em algum nó com necessidades iguais, ou maiores a do nó anterior. Utiliza *soft states* para que seja possível reconfigurar a reserva no caso de uma mudança de rota, ou seja, uma mudança no circuito criado. O *soft state* é alterado, quando necessário, por meio de mensagens de atualização periódicas. A ausência de pacotes de atualização provoca um expiramento na reserva, que então é liberada.

2.3.1.1 Visão Geral da Operação do RSVP

No *RSVP*, os roteadores devem ser capazes de reservar recursos a fim de fornecer *QoS* especial para fluxos de pacotes específicos do usuário. Assim, os roteadores *RSVP* devem possuir 4 (quatro) componentes, como mostrado pela Figura 2.1 (Melo, 2001).

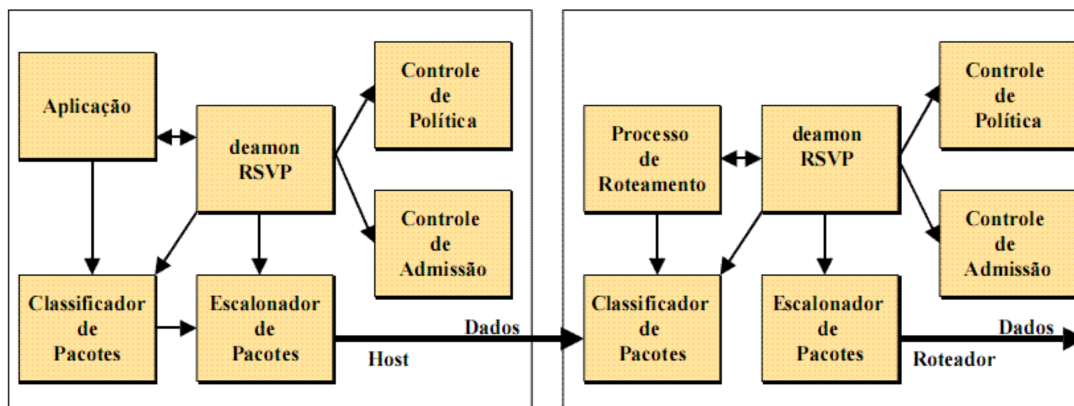


Figura 2.1: Host e Roteador com Suporte a RSVP.

Fonte: Melo (2001).

Protocolo de Sinalização: aplicações necessitando de *QoS* devem configurar um caminho e reservar recursos antes da transmissão de seus dados. Para isto elas devem usar um protocolo de sinalização, neste caso o *RSVP*.

Controle de Admissão: verifica se um pedido de alocação de recursos pode ser garantido. Ela implementa o algoritmo de decisão que um roteador ou *host* usa para determinar se um novo fluxo pode ter sua *QoS* garantida sem afetar fluxos já garantidos.

Classificador: quando um roteador recebe um pacote, o classificador executa uma classificação *MultiField classifier (MF)* e coloca o pacote em uma fila específica em função do resultado da classificação. A classificação *MF* é o processo de classificar pacotes, baseado no conteúdo dos seus campos tais como: endereço fonte, endereço destino, *Tipo de Serviço (TOS)* e identificador do protocolo.

Cada pacote que chega deve ser mapeado em alguma classe; todos os pacotes na mesma classe obtêm o mesmo tratamento do escalonador. Uma classe pode corresponder a uma grande categoria de fluxos. Uma classe é uma abstração que pode ser local a um roteador particular. O mesmo pacote pode ser classificado de várias formas por diferentes roteadores ao longo do caminho.

Escalonador: após a classificação, o escalonador seleciona para transmissão o pacote de modo a satisfazer os requisitos de *QoS*. O escalonador de pacotes gerencia a retransmissão dos diferentes pacotes usando um conjunto de fila.

O *RSVP* trabalha com 2 (dois) tipos principais de mensagens:

PATH: mensagens enviadas periodicamente pelo transmissor ao endereço multicast. Contém a especificação de fluxo (formato de dados, endereço fonte, porta fonte) e características de tráfego. Essa informação é utilizada pelos receptores para achar o caminho reverso ao transmissor e determinar quais recursos devem ser reservados. Os receptores devem se cadastrar no grupo multicast a fim de receber mensagens *PATH*;

RESV: mensagens geradas pelos receptores contendo parâmetros de reserva, como especificação de fluxo e de filtro. O filtro determina quais pacotes no fluxo de dados devem ser usados no classificador de pacotes. A especificação de fluxo é usada no escalonador, que procura manter a necessidade do receptor.

A operação do *RSVP* envolve a troca desses dois tipos de mensagens entre *hosts* e roteadores ao longo de um fluxo. A Figura 2.2 ilustra a troca dessas mensagens ao longo do caminho de um fluxo.

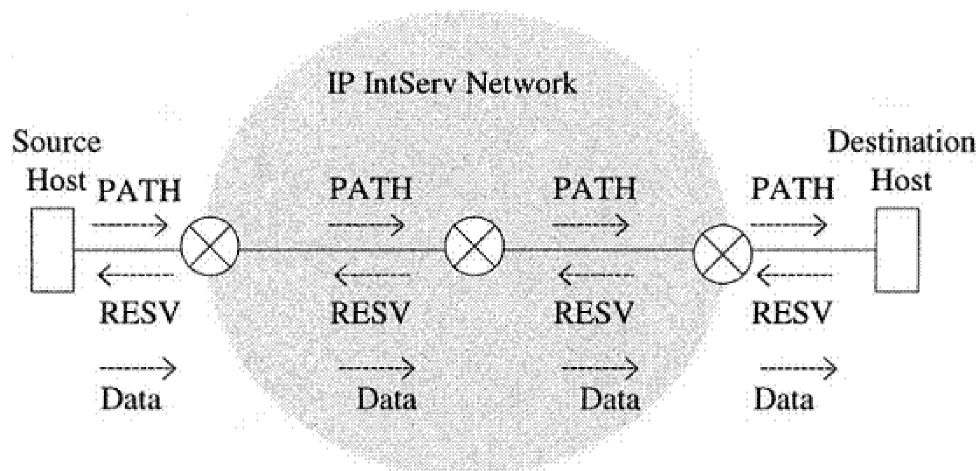


Figura 2.2: Operação RSVP.

Fonte: Park (2005).

O *host* emissor envia uma mensagem de *PATH* para o *host* destino para um fluxo ou uma sessão. A mensagem de *PATH* contém a especificação de fluxo para um fluxo. Assim que a mensagem *PATH* passa pelos roteadores ao longo do caminho, os roteadores registram a identificação e a especificação de fluxo. Quando a mensagem *RESV* correspondente chega de um *host* receptor, os roteadores fazem a correlação necessária entre a informação contida na mensagem de *PATH* e a mensagem *RESV*. Quando os receptores recebem a mensagem *PATH*, são enviadas as mensagens *RESV*. As mensagens de *RESV* carregam informações sobre a reserva de recursos. Os pacotes *IP* trafegam na direção especificada pela mensagem *PATH*.

2.4 Serviços Diferenciados

A arquitetura de Serviços Diferenciados posiciona-se entre os extremos do *best-effort* e serviços integrados. Na arquitetura não existe alocação explícita de recursos e não é feita sinalização, permitindo maior escalabilidade e menor sobrecarga com sinalização.

Atualmente, a diferenciação de serviços é implementada com sucesso por meio do *DiffServ*, assunto da próxima seção.

2.4.1 DiffServ

O *DiffServ* oferece um esquema para prover várias classes de diferenciação de serviços para tráfegos em redes. Sua arquitetura propõe uma variedade de serviços, sendo que cada pacote contém informações necessárias para que os roteadores, ao longo do caminho, saibam qual tratamento deve ser dado ao pacote durante o encaminhamento. Os serviços que podem ser aplicados a pacotes de um fluxo são definidos por um contrato junto ao *Provedor de Serviços de Internet (ISP)* do usuário, o *Contrato de Nível de Serviço (SLA).*

O esquema do *DiffServ* trata-se de uma alteração no protocolo *IP*, em que o antigo campo *TOS* do cabeçalho do protocolo *IP* passa a ser chamado *DS* e a carregar as características de serviço do pacote. O campo *DS* determina qual será o tratamento dado ao pacote de acordo com a classe de *QoS*, ou seja, por meio do valor estipulado neste campo é possível que o roteador saiba quais são as prioridades atribuídas ao pacote. A Figura 2.3 mostra o cabeçalho do protocolo *IP* versão 4 (quatro) com o campo *DS* em destaque.

Na arquitetura *DiffServ* não existe alocação explícita de recursos e não é feita sinalização, tendo em vista que a prioridade do pacote é transmitida no cabeçalho *IP*, permitindo desta forma maior escalabilidade e baixa sobrecarga de sinalização.

A idéia fundamental do *DiffServ* é definir um conjunto de mecanismos implementados nos nós da rede (*hosts* e roteadores) que suportem uma grande variedade de serviços. Esses serviços são oferecidos no interior de um domínio *DiffServ* que é composto por um conjunto de nós que compartilham uma mesma política de serviços. A Figura 2.4 mostra um esquema de domínio *DiffServ* destacando os componentes de borda.

Os nós de borda de um domínio *DiffServ* são responsáveis pela classificação e condicionamento do tráfego que entra no domínio. Para cada fluxo de tráfego entrando pelos nós de borda, a política de *QoS* define qual tem direito um serviço diferenciado, como

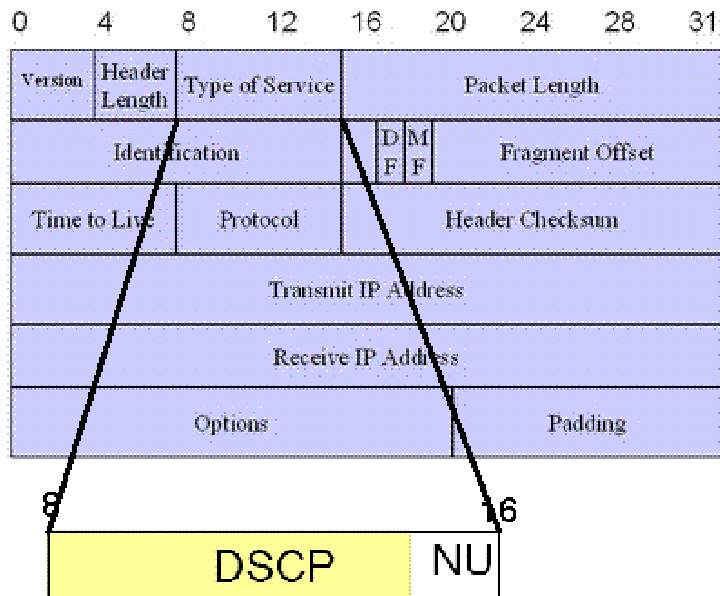


Figura 2.3: Cabeçalho IPv4.

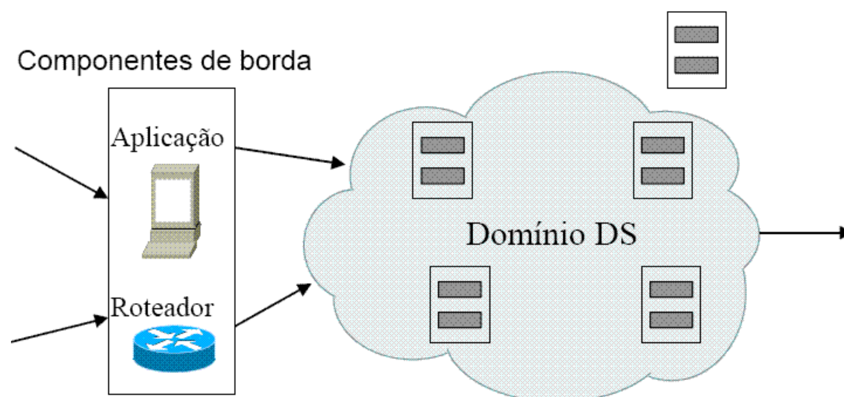


Figura 2.4: Esquema de um Domínio DiffServ.

Fonte: Melo (2001).

este deve ser marcado nos nós de borda e como deve tratado pelos nós interiores. Os nós interiores examinam a marcação dos pacotes e atuam de acordo com as políticas definidas ou seu perfil de tráfego.

O *DiffServ* define um número de classes de serviço e mecanismos de *QoS*, conhecidos por *Comportamento por Host (PHB)*, que são aplicados aos pacotes nas classes de serviço. Os *PHBs* são comportamentos que devem estar presentes nos nós de uma rede para fornecer serviços diferenciados a um pacote. Na Figura 2.3 os 6 (seis) primeiros *bits* correspondem ao *Differentiated Services Code Point (DSCP)*, enquanto os 2 (dois) últimos *bits* são não utilizáveis e podem ser usados para definições futuras de *PHBs*. O campo *DSCP* é utilizado pelos nós internos para saber qual comportamento deve ser aplicado ao pacote.

Dentre os padrões de *PHB* estão em destaque o *Expedited Forwarding (EF)* e o *Encaminhamento Garantido (AF)*.

O *EF* tem como princípio básico diminuir o tempo de permanência em filas dos pacotes em trânsito. Este deve prover baixa latência e serviço de alta prioridade, sendo ideal para aplicações intolerantes a atrasos. Os pacotes devem ter a mais alta prioridade de emissão e a mais baixa prioridade de descarte. Para alcançar este comportamento, todos os nós da rede devem garantir que o tráfego *EF* tem probabilidade mínima de atraso, *jitter*, perdas. Uma vez que o serviço tenta emular um linha dedicada sobre a rede *IP*.

O *AF* não fornece garantia uma garantia estrita, mas apenas uma expectativa de serviço quando a rede passar por momentos de congestionamento. Neste comportamento, os pacotes podem ser descartados em momentos de muito congestionamento e saturação de armazenamento das filas. O comportamento *AF* consiste de 4 (quatro) classes diferentes de serviços, cada uma com 3 (três) níveis de prioridade de descarte diferentes. Resultando em 12 valores diferentes de *DSCP*. Os roteadores usam os valores de precedência de descartes para determinar quem descartar em caso de congestionamento. A Tabela 2.1 mostra as diferentes classes definidas por *AF*.

Tabela 2.1: Classes de serviço para comportamento de encaminhamento garantido

Prioridade de Descarte	Classes AF			
	Classe 1	**Classe 2**	**Classe 3**	**Classe 4**
Baixa	AF11	AF12	AF13	AF14
Média	AF21	AF22	AF23	AF24
Alta	AF31	AF32	AF33	AF34

Dois importantes componentes são definidos na arquitetura DiffServ como mostrado pela Figura 2.5 que ilustra um roteador na borda de um domínio. Esses componentes são: o componente de classificação e o componente de condicionamento de tráfego.

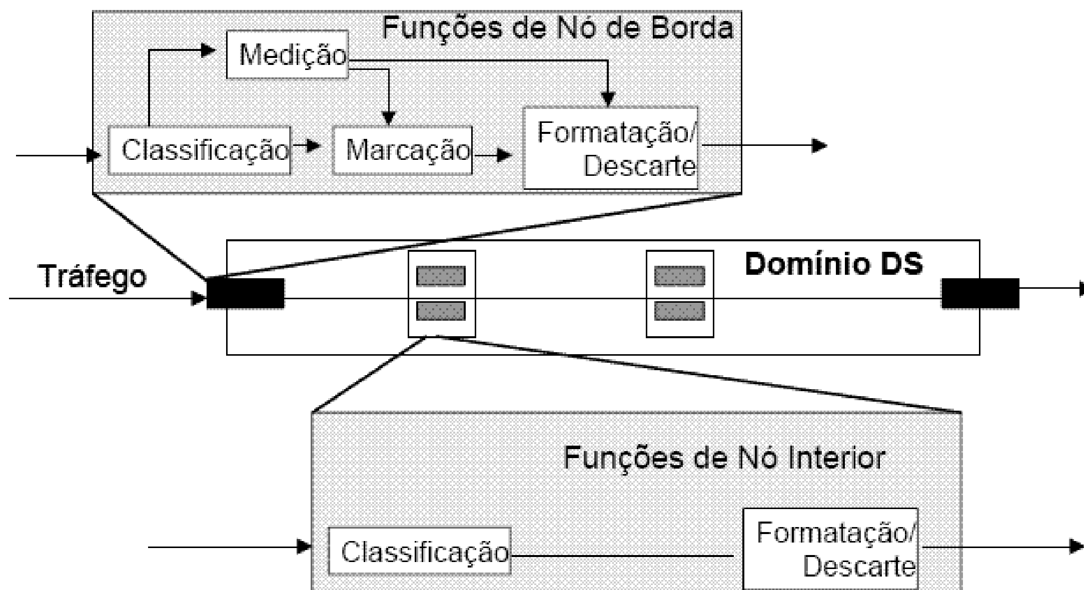


Figura 2.5: Roteador com Suporte a DiffServ.

Fonte: Melo (2001).

São dois os classificadores existentes: um que classifica o fluxo baseado apenas na classificação *DS*, classificador de comportamento agregado ou *Behaviour Agregate (BA)*, e outro que verifica múltiplos campos no cabeçalho *IP*, o classificador de multi-campo ou *MF*.

O classificador é o componente que divide o fluxo de entrada em um conjunto de fluxos de saída por meio de filtros de tráfego baseados no conteúdo do cabeçalho do pacote e em diferentes atributos do pacote que podem ser implicitamente derivados.

O medidor verifica se o pacote que chega está de acordo com um perfil de tráfego pré-definido. Com base nessa nessa comparação podem ser executadas 3 (três) ações, que podem ou não serem combinadas: marcação, formatação e descarte.

O escalonamento é o processo de decidir qual fila, dentre as candidatas, deve ser servida durante o processo de transmissão. O escalonador deve ser usado quando os recursos de saída não são suficientes para o tráfego que entra. A formatação modifica o tráfego de entrada para forçar um determinado perfil de saída.

Os condicionadores de tráfego são empregados em um determinado estágio do caminho dos dados para forçar uma determinada política, e são implementados por meio de combinação de um ou mais componentes de *DiffServ*.

Capítulo 3

Fundamentos de Computação Distribuída

Tanenbaum e Steen (2002) apresenta a seguinte definição para sistemas distribuídos: "Um sistema distribuído é uma coleção de computadores independentes que aparenta, ao usuário do sistema, ser um único computador". Em outras palavras, essa definição reflete que conjuntos de computadores autônomos, possivelmente heterogêneos, comunicam-se e realizam suas tarefas de maneira transparente ao usuário.

A principal diferença entre sistemas distribuídos e sistemas centralizados é a comunicação interprocesso, ou seja, a forma com que os processos trocam informações durante suas execuções. Uma vez que sistemas distribuídos são formados por computadores individuais e autônomos, qualquer tipo de comunicação interprocesso por memória compartilhada é impossível. Logo, os processos comunicam-se através de uma rede de comunicação.

Existem diversos padrões que permitem a troca de mensagens entre processos através de uma rede de maneira transparente. As próximas seções apresentam alguns desses padrões, começando por *RPC*, um padrão baseado no modelo cliente-servidor, que tem por característica não alterar a forma com que os desenvolvedores realizam chamadas aos procedimentos. As demais seções apresentam padrões de objetos e componentes distribuídos, tais como: *DCOM*, *CORBA* e *RMI*, que têm a *RPC* por base.

3.1 RPC

O objetivo da *RPC* é permitir a comunicação interprocesso entre máquinas distintas de maneira transparente ao usuário. Essa comunicação é feita por meio de chamadas remotas de procedimentos, nas quais uma máquina - o cliente - requisita a execução de um procedimento em outra máquina - o servidor - utilizando de mensagens transmitidas através da rede de comunicação.

RPC foi projetada de forma que não exista diferença entre uma chamada de procedimento local e uma chamada de procedimento remota, poupando o desenvolvedor de qualquer aprendizado adicional para desenvolvimento de aplicações com *RPC*. Um novo nível de abstração é inserido no desenvolvimento, escondendo todos os detalhes de comunicação, tais como localização, montagem e desmontagem de mensagens, entre outros aspectos. As próximas subseções apresentam os aspectos de projeto de *RPC*, bem como descrevem como acontece sua operação.

3.1.1 Aspectos de Projeto de RPC

Embora as chamadas locais e remotas sejam idênticas, a execução de uma *RPC* é bastante diferente de uma execução local, a começar pela possibilidade de *crash* de uma das máquinas envolvidas, que é a interrupção da comunicação e a perda de mensagens, implicando tanto em falha na execução do procedimento remoto, quanto na perda da transparência de localização do procedimento.

O projeto de *RPC* considera outros aspectos de implementação como a passagem de parâmetros, por exemplo. Uma vez que a chamada e a execução acontecem em máquinas autônomas, é impossível passar parâmetros por referência. Para corrigir esse problema surgiu o método copia-restaura, no qual o valor do parâmetro é copiado do cliente para o servidor, e novamente copiado para o cliente sobrescrevendo o valor anterior após a execução. Embora o método copia-restaura consiga o mesmo resultado de uma passagem

de parâmetro por referência, o mesmo implica em uma sobrecarga na comunicação conseqüente da cópia de volta para o cliente, além de uma inevitável latência custo das diversas cópias de valores que ocorrem durante a execução do procedimento.

Outro aspecto considerado é a heterogeneidade do conjunto de arquiteturas, que pode incorrer em representações de tipos de dados incorretas. Esse problema é decorrente da comunicação entre máquinas com representações de dados diferentes, como por exemplo, computadores que utilizam numeração de *bytes* da direita para esquerda, padrão conhecido com *little-endian*, e computadores com numeração da esquerda para direita, ou *big-endian*. Tal problema é corrigido com a adição de um *byte* indicando qual o formato de dados utilizado pelo emissor na mensagem transmitida, sendo que qualquer modificação necessária de representação fica sob responsabilidade do receptor da mensagem. A adição desse *byte* evita a necessidade de um tipo de dados padrão à rede, para o qual todas as máquinas teriam de converter seus dados a serem transmitidos. Essa adição resultaria em uma latência indesejável caso duas máquinas, de mesma arquitetura, tivessem que converter seus dados para a representação padrão ao invés de transmiti-los normalmente.

3.1.2 Operação em RPC

As chamadas remotas de procedimento são feitas da mesma forma que as chamadas locais. Dessa forma, a assinatura de procedimento utilizada pelo cliente deve ser a mesma do procedimento que o servidor executa.

Essas assinaturas são agrupadas em conjuntos que constituem as especificações formais dos servidores, também conhecidas por interfaces. Tais especificações contêm descrições que definem particularidades dos procedimentos como nome, tipo de retorno, lista e tipos de parâmetros, além de indicar se cada um dos parâmetros é de entrada, saída ou ambos. A especificação formal do servidor contém também o nome e a versão do mesmo, sendo essas informações utilizadas para registrar o servidor no *binder*.

O *binder* é um programa no qual todo servidor ativo está registrado. Sempre que um servidor torna-se ativo, sua interface é exportada para o *binder* por meio do registro. Após o registro, clientes podem importar a interface do servidor por meio de uma consulta ao *binder* e, assim, fazer chamadas remotas de procedimento diretas ao servidor específico.

Um conceito criado por *RPC* é o de procedimentos *stubs*, que são os principais responsáveis pela transparência na comunicação interprocesso. Esses procedimentos agem como substitutos, interceptando a chamada e o retorno do procedimento e encaminhando-os para os lugares adequados. Existem dois tipos de procedimentos *stubs*: o cliente *stub* e o servidor *stub*. O cliente *stub* localiza-se no cliente, sendo o responsável por receber a chamada e encaminha-lá ao servidor, enquanto que o servidor *stub* localiza-se no servidor e tem a responsabilidade de receber o retorno da execução e encaminhá-lo de volta ao cliente. Outras tarefas como a importação de interfaces adequadas no *binder* e a montagem de mensagens que trafegam pela rede também estão a cargo dos *stubs*.

A montagem de mensagem, ou *marshalling*, envolve a preparação e empacotamento dos parâmetros, e o encaminhamento para o servidor correto, enquanto que a desmontagem, ou *unmarshalling*, ainda trata das conversões para as representações de tipos de dados necessárias. A Figura 3.1 mostra os passos percorridos por uma *RPC* desde a chamada até o retorno.

Na Figura 3.1 são identificados os seguintes passos:

1. O *cliente* faz a chamada de procedimento ao *cliente stub*.
2. Por sua vez, o *cliente stub* realiza o *marshalling* dos parâmetros e encaminha a mensagem ao *servidor stub*.
3. Ao receber a mensagem do *cliente stub*, o *servidor stub* realiza o *unmarshalling* da mensagem, e faz um chamada local para o *servidor*.
4. O *servidor* executa o procedimento e devolve os resultados ao *servidor stub*.

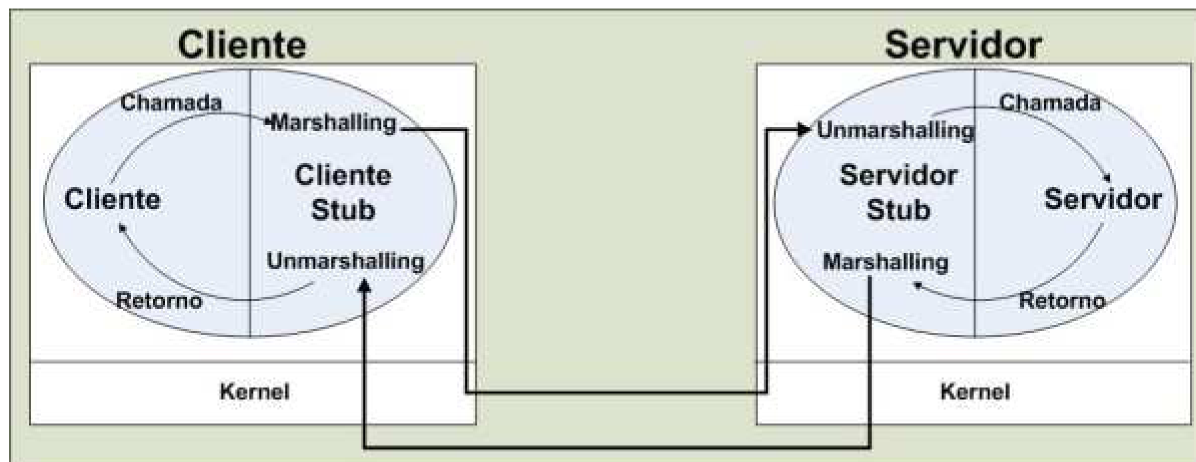


Figura 3.1: Estrutura de uma RPC.

Fonte: Tanenbaum e Steen (2002)

5. O *servidor stub* faz o *marshalling* dos resultados e encaminha ao *cliente stub*.

6. Ao receber o mensagem do *servidor stub*, o *cliente stub* realiza o *unmarshalling* dos resultados, entregando o retorno ao *cliente*.

RPC é uma maneira elegante de lidar com a comunicação interprocesso entre máquinas autônomas. No entanto, outros padrões têm surgido e considerado novos paradigmas de programação como: a programação orientada a objetos e a programação orientada a componentes. Tais padrões ainda baseiam-se na arquitetura *RPC*, mas suportam novas tecnologias sempre com o intuito de facilitar o trabalho do desenvolvedor. As próximas seções abordam com detalhes algumas dessas novas arquiteturas mais populares.

3.2 DCOM

DCOM é uma evolução sem costura da arquitetura de software *Component Object Model (COM)* (Willliams e Kindel, 1994) desenvolvida pela *Microsoft* para criação de componentes reutilizáveis.

COM define um padrão de interoperabilidade de software extensível, independente de plataforma e linguagem de programação, permitindo a comunicação entre componentes de diferentes fornecedores. Essa comunicação está limitada apenas a aplicações que coexistam em uma mesma máquina, sendo possibilitada por meio da exposição de interfaces, utilizadas por uma aplicação, para acessar componentes de outra como pode ser observado na Figura 3.2.

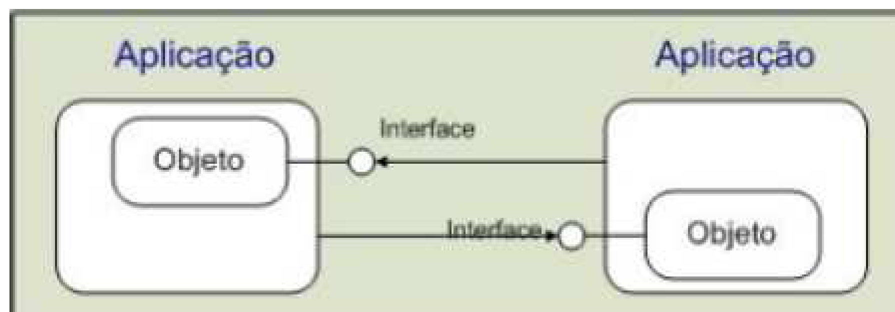


Figura 3.2: Comunicação de Componentes Entre Processos de Uma Mesma Máquina.

Um cliente *COM* pode conectar-se com um ou mais objetos que estão em servidores *COM*, sendo um cliente *COM* qualquer aplicativo que queira utilizar um serviço fornecido por algum dos objetos (Redmond, 1997). Os servidores *COM* são binários que contêm implementações de objetos.

Os objetos e interfaces *COM* devem ser identificados universalmente de maneira única, ou seja, deve existir um e apenas um objeto ou interface identificado com um dado valor. Isso é possível por meio do uso de *Globally Unique Identifier (GUID)*, que são identificadores únicos globais de 128 *bits* (Rubin e Brain, 1999). A Figura 3.3 ilustra a estrutura do *GUID*.

Em *COM*, toda iniciativa deve partir do cliente. O servidor é uma entidade passiva que apenas responde às requisições dos clientes. A Tabela 3.1 mostra uma típica interação entre cliente e servidor.

A interação componente-cliente *COM* foi definida para ocorrer de maneira transparente. Uma vez que um cliente não pode acessar diretamente um componente de outra apli-

```
typedef struct _GUID
{
  unsigned long Data1;
  unsigned short Data2;
  unsigned short Data3;
  unsigned char Data4[8];
} GUID;
```

Figura 3.3: Estrutura de GUID.

Tabela 3.1: Interação Típica Entre Cliente e Servidor COM.

Requisição do Cliente	**Resposta do Servidor**
Faz requisição de acesso a uma interface COM, especificando a classe e a interface COM (pelo GUID)	Inicia o servidor, se necessário. Cria o objeto COM requisitado. Cria uma interface para o objeto COM. Incrementa o contador de referências das interfaces ativas e retorna ao cliente.
Chama os métodos da interface	Executa os métodos em objetos COM.
Libera a interface	Decrementa o contador de referências da interface. Deleta o objeto se o contador for zero. Se não há mais conexões com o servidor, então desativa o servidor.

Fonte: Rubin e Brain (1999)

cação, *COM* fornece um meio para comunicação interprocesso, garantindo a transparência por meio da interceptação da chamada do cliente e o encaminhamento para o componente de outro processo. A Figura 3.4 ilustra com as bibliotecas *COM run-time* permitem a interação componente-cliente.

A Figura 3.4 ilustra a comunicação entre um cliente e um objeto localizados em processos distintos. A implementação dos objetos dentro dos limites físicos de uma mesma máquina ou fora desses limites é discutida com maiores detalhes na próxima subseção.

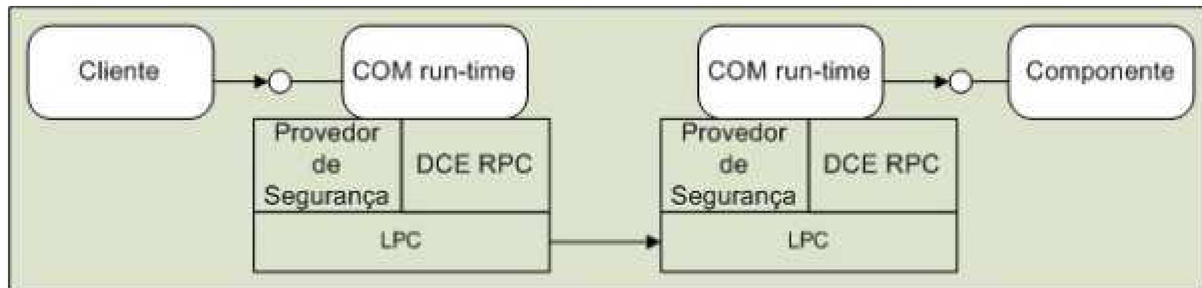


Figura 3.4: Interação Cliente-Componente Utilizando COM.

3.2.1 Trasparência de Localização

Os objetos *COM* podem estar implementados local ou remotamente em relação ao cliente. No caso de residirem na mesma máquina que o cliente, ainda existem 2 possíveis situações de implementação: *in-process* e *out-process*. A implementação *in-process* é caracterizada pela implementação de uma *Dynamic Link Library (DLL)* carregada diretamente no espaço de endereçamento de memória do processo cliente, enquanto que na implementação *out-process* um binário executável, com espaços próprios de endereçamento de memória, age como servidor (Redmond, 1997).

Para que o desenvolvedor da aplicação cliente não tenha de se preocupar com o tipo de implementação do servidor, ou seja, a localização do servidor, *COM* fornece um modelo de programação único provê transparência de localização. Isso só é possível graças ao uso de interfaces, as quais são, na verdade, ponteiros para uma tabela virtual de funções, a qual possui ponteiros para as funções verdadeiras (Redmond, 1997).

Dessa forma, quando um servidor é implementado *in-process* a tabela virtual de funções aponta para métodos que estão localizados diretamente no espaço de endereçamento de memória do cliente. No entanto, quando o servidor é implementado *out-process*, este possui uma área própria de endereçamento de memória para a qual a tabela não pode apontar. Esse problema é resolvido por meio da utilização de um *proxy*. O *proxy*, localizado no espaço de endereçamento do cliente, existe para direcionar as chamadas para um outro elemento localizado na área de endereçamento do servidor, o *stub*. *Proxies COM* e *stubs*

COM funcionam de maneira equivalente aos mostrados na Figura 3.1, sendo responsáveis pelas mesmas funções de *marshalling* e *unmarshalling*.

Quando o servidor é implementado remotamente, a comunicação interprocesso acontece da mesma maneira que a implementação *out-process*, no entanto, servidor e cliente encontram-se em máquinas diferentes. Esse é o princípio de *DCOM*: permitir que clientes acessem objetos *COM* localizados em diferentes máquinas que se comunicam por através de uma rede de comunicação.

3.2.2 Arquitetura DCOM

DCOM pode ser entendido como uma extensão de *COM* que permite que objetos *COM* sejam distribuídos por uma rede de comunicação. *DCOM* é construído sobre a implementação de *RPC*, sendo assim, é transparente a *DCOM* quais são os protocolos que compõem o restante da pilha abaixo de sua implementação.

DCOM expande a arquitetura *COM* mostrada na Figura 3.4 permitindo que os componentes comuniquem-se através de uma rede de comunicação, utilizando *RPC* para enviar e receber informações entre componentes. Dessa forma, *DCOM* manipula os detalhes de baixo nível dos protocolos de comunicação, permitindo que o desenvolvedor ocupe-se apenas com a implementação das regras de negócio.

Para permitir a comunicação cliente-componente localizados em máquinas diferentes, *DCOM* substitui a comunicação local interprocesso, realizada por *Chamada de Procedimento Local (LPC)*, por uma pilha de protocolos de comunicação, mantendo a comunicação transparente. A Figura 3.5 mostra a alteração na arquitetura ilustrada na Figura 3.4.

DCOM serve para os mesmo propósitos de outras arquiteturas, como a arquitetura *CORBA*, assunto da próxima seção.

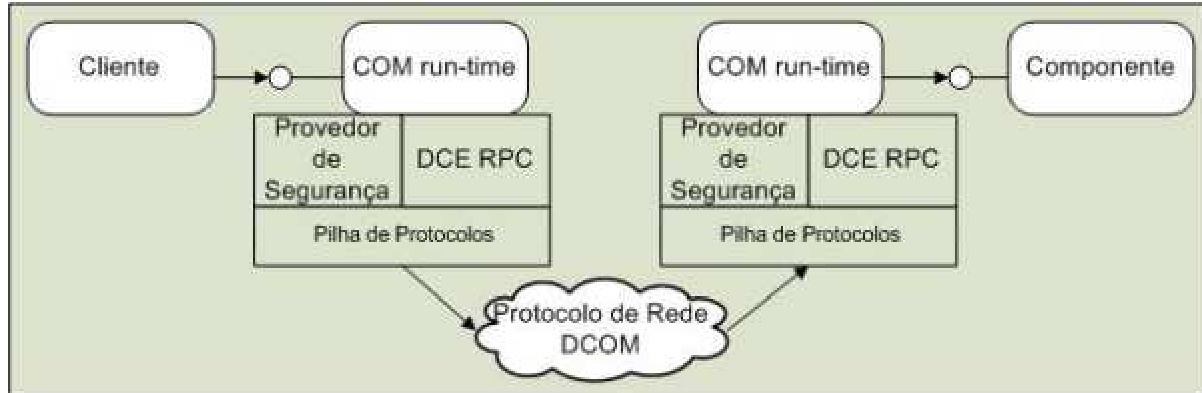


Figura 3.5: Interação Cliente-Componente Utilizando DCOM.

3.3 CORBA

CORBA é uma arquitetura padrão de responsabilidade do *Object Management Group (OMG)* (OMG, 2006) para sistemas de objetos distribuídos, que permite interoperabilidade entre coleções dos mesmos, por meio de automação de tarefas de baixo nível e de comunicação como:

- Registro, localização e ativação de objetos;
- Preparação e demultiplexação de requisições;
- Tratamento de erros;
- Preparação de parâmetros;
- Despachamento de operações.

CORBA é uma das primeiras especificações adotadas pelo *OMG* para *Object Management Architecture (OMA)* (Stone, 1995), cuja idéia é disponibilizar um serviço distribuído chamado *Object Request Broker (ORB)* responsável por todos mecanismos necessários para encontrar, instanciar e executar um objeto. A *OMA* é uma infraestrutura conceitual sob a qual todas as especificações da *OMG* devem ser construídas (omg, 1999). A Figura 3.6 ilustra o modelo de referência *OMA*.

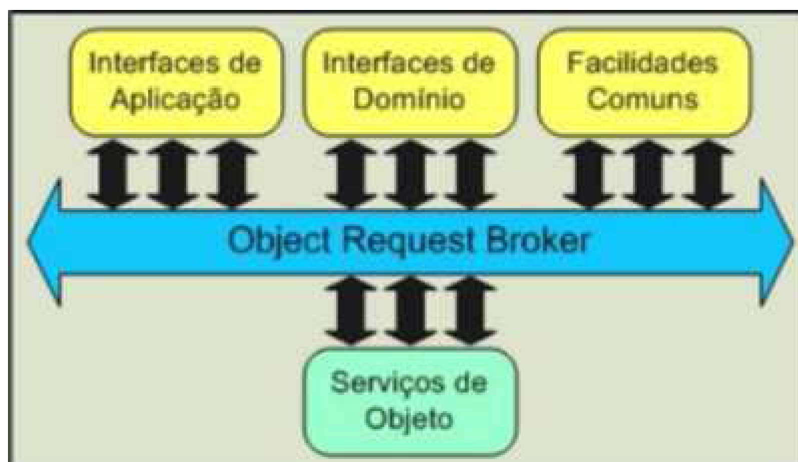


Figura 3.6: Arquitetura do Modelo de Referência OMA.

Vinoski (1997) define os 4 componentes dessa arquitetura, ilustrada na Figura 3.6, da seguinte forma:

- **Serviços de Objeto**: São interfaces independentes de domínio que são utilizadas por muitas aplicações distribuídas. Um exemplo de serviço que utiliza essas interfaces é o serviço de descoberta de objetos, o qual permite ao cliente encontrar objetos com base no nome ou nas propriedades dos mesmos. Além desse, serviços de segurança, gerenciamento de ciclo de vida de objetos, entre outros, também fazem uso dessas interfaces.

- **Facilidades Comuns**: São bastante semelhantes às interfaces de Serviço de Objetos, no entanto, são mais voltadas para as aplicações de usuário final.

- **Interfaces de Domínio**: Semelhantes às interfaces de Serviços de Objeto e Facilidades Comuns, porém são comuns a aplicações de um determinado domínio. Uma vez que uma Interface de Aplicação é utilizada por muitas aplicações do mesmo domínio, esta pode ser promovida a Interface de Domínio.

- **Interfaces de Aplicações**: São interfaces desenvolvidas para uma aplicação específica.

O núcleo do modelo de referência mostrado na Figura 3.6, no caso o *ORB*, permitem que objetos façam requisições e recebam respostas de maneira transparente em um ambiente distribuído. O *ORB* constitui o elemento base para construção de aplicações de objetos distribuídos entre aplicações e ambientes heterogêneos.

Detalhes das interfaces e características do componente *ORB* são detalhados por *CORBA*. A 3.7 ilustra a arquitetura *CORBA* e seus componentes podem ser conceituados como:

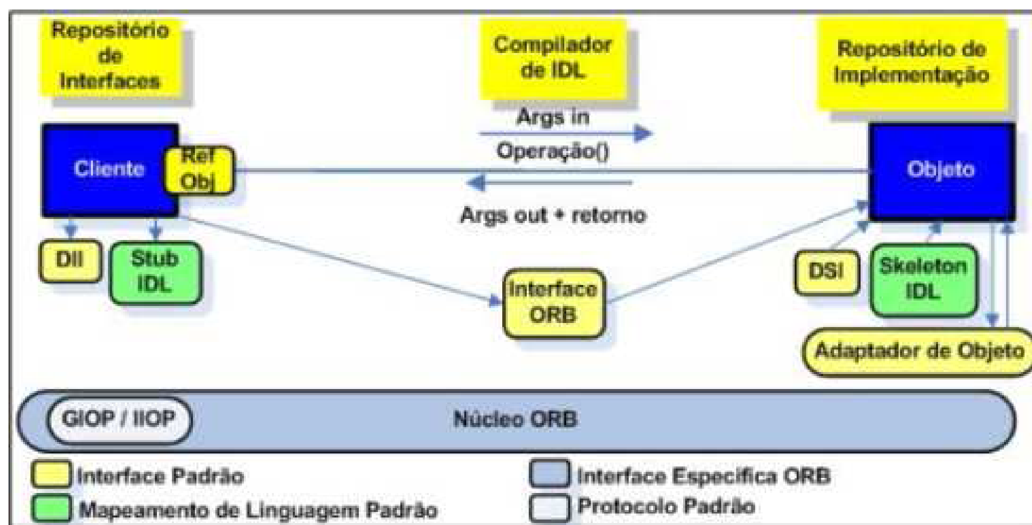


Figura 3.7: Arquitetura CORBA.

- **Cliente**: Componente que requisita uma operação de um determinado objeto. Para o cliente, a chamada ao objeto é transparente e realizada tal qual uma chamada local. O cliente não conhece a localização, implementação e estado do objeto, além de desconhecer os mecanismos de comunicação utilizados (*e.g. Transmission Control Protocol (TCP) / IP*, memória compartilhada, método local) na chamada do objeto.

- **Objeto**: Componente da arquitetura que fornece operações por meio de interfaces.

- **Núcleo *ORB***: Responsável pela transparência da interação entre cliente e objeto. Basicamente, a funcionalidade desse componente está em entregar requisições ao objeto e o retorno ao cliente.

- **Interface *ORB***: Uma vez que *ORB* pode ser implementado de diferentes maneiras e em diferentes linguagens de programação, *CORBA* define uma interface abstrata para *ORB*, separando os serviços da implementação.

- ***Stubs* e *Skeletons Interface Definition Language (IDL)***: Funcionam como substitutos em relação ao cliente e objeto reais, respectivamente. *Stub* e *Skeletons IDL* são criados a partir de definições formais fornecidas pela especificação das interfaces em *IDL*, e são responsáveis pelas operações de *marshalling* e *unmarshalling* dos dados trocados na interação cliente-objeto.

  Do lado do cliente, quando este faz um requisição, o *Stub* intercepta essa requisição, organiza os parâmetros e envia ao *Skeleton*. Do lado do objeto, quando chega a requisição ao *Skeleton*, este faz uma requisição local ao objeto, passando os parâmetros recebidos e aguarda retorno. Quando o objeto retorna ao *Skeleton*, este organiza o resultado e envia ao *Stub*, o qual entrega o retorno ao cliente. Quando o objeto retorna ao *Skeleton*, este organiza o resultado e envia ao *Stub*, o qual entrega o retorno ao cliente.

- ***Dynamic Invocation Interface (DII)* e *Dynamic Skeleton Interface (DSI)***: São similares ao *Stub* e *Skeleton*, respectivamente. No entanto, são dinâmicos e não requerem substitutos de interfaces específicas em *IDL*.

- **Adaptador de Objeto**: Associa a implementação de um objeto ao *ORB*, auxiliando-o na entrega de requisições e ativação de objetos.

3.3.1 Definição de Interfaces

Antes de requisitar uma operação de um objeto, uma aplicação deve conhecer a interface do objeto. A interface especifica a operação, os tipos suportados e as requisições que podem ser feitas ao objeto. Interfaces são escritas em *OMG IDL*, uma linguagem declarativa que separa as assinaturas da implementação das operações.

A definição de interfaces para objetos pode acontecer de duas maneiras diferentes que, no entanto, são equivalentes. A primeira maneira utiliza a linguagem de definição de interfaces *OMG IDL*, enquanto a segunda maneira adicioná-las ao serviço de repositório de interfaces. Este serviço representa os componentes de uma interface como objetos que podem ser acessados em tempo de execução.

Além de permitir que clientes acessem objetos remotamente, as interfaces têm outra papel fundamental em *CORBA*, uma vez que a geração de *stubs* e *skeletons* está diretamente ligada às interfaces, como mostrado na Figura 3.8.

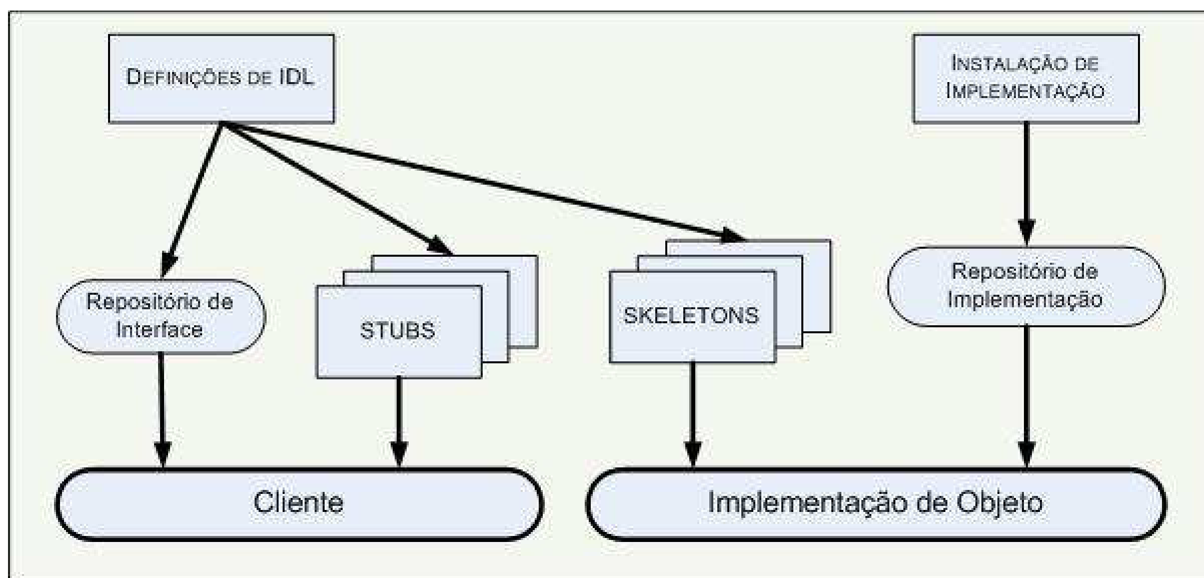


Figura 3.8: Repositório de Interfaces e Implementação.

A Figura 3.8 também ilustra o repositório de implementação, residente no cliente. Esse repositório é criado durante a instalação dos objetos, sendo utilizado sempre que uma requisição é direcionada a algum objeto.

3.3.2 Implementação de Clientes e Objetos

Os clientes possuem referências com as quais podem invocar um objeto remoto. A invocação de um objeto pode ser divididas em várias tarefas como:

- Especificação o objeto a ser invocado;
- Definição da operação a ser realizada pelo objeto;
- Definição dos parâmetros devem ser enviados ao objeto, sejam eles de entrada, saída ou ambos.

De modo geral, o *ORB* é responsável por todo trabalho de baixo nível na comunicação e na transferência dos dados entre o cliente e o objeto, fornecendo total transparência a ambos, a menos que algum falha inesperada ocorra durante a invocação. A transparência concentra-se em não modificar o modo como o cliente invoca objetos remotos e locais.

Para invocar um objeto remoto, o cliente deve acessar *stubs* específicos de um tipo de objetos como qualquer rotina de sua biblioteca. Para o cliente a chamada ocorre como qualquer chamada local, no entanto os *stubs* interagem com o *ORB* para efetiva invocação do objeto. A Figura 3.9 mostra uma estrutura típica de um programa cliente.

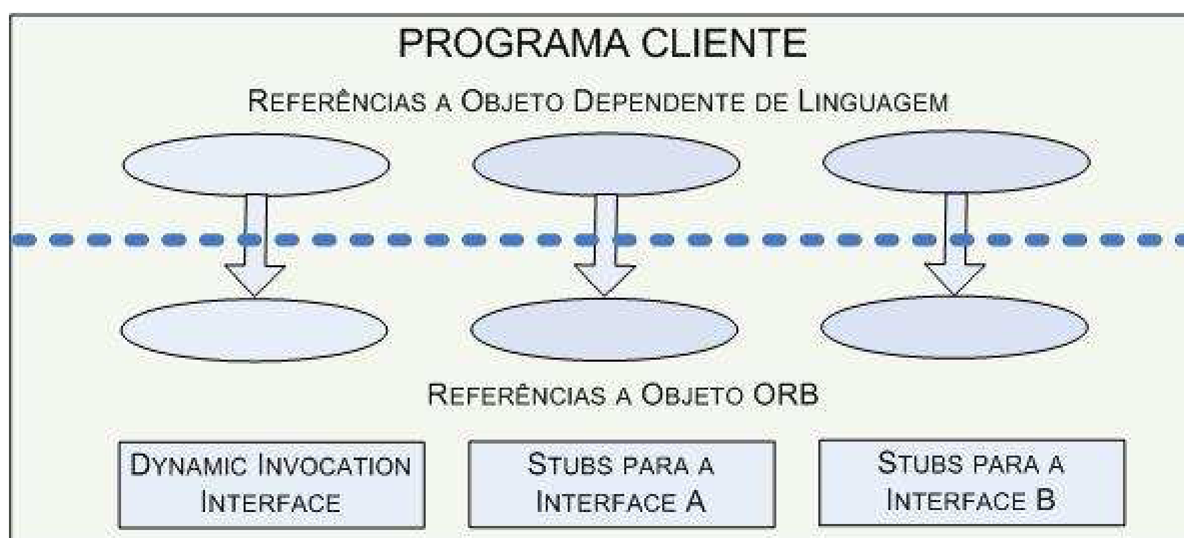


Figura 3.9: Implementação Típica de Cliente CORBA.

A implementação do objeto fornece o comportamento e o estado atual de um objeto (omg, 1999). As implementações de objetos interagem com o *ORB* através do *Object Adapter*, o qual fornece uma interface para serviços *ORB* que são convenientes para um

estilo particular de implementação de objetos. A Figura 3.10 ilustra uma implementação típica de um objeto *CORBA*.

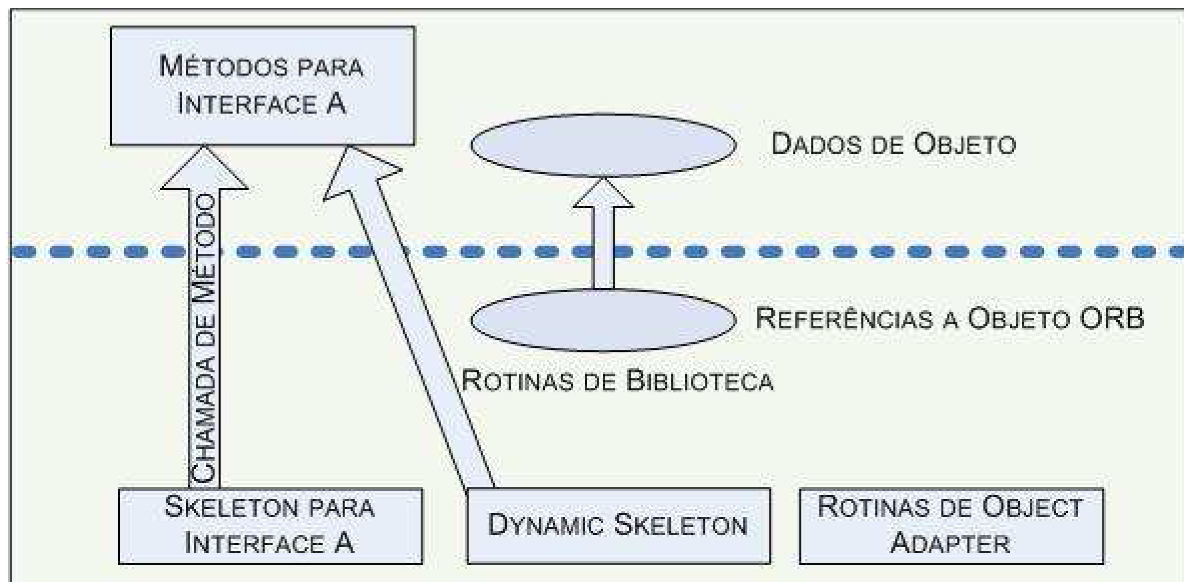


Figura 3.10: Implementação Típica de Objeto CORBA.

3.4 RMI

Java RMI, ou simplesmente *RMI*, permite a criação de aplicações distribuídas baseadas em tecnologia Java, nas quais métodos de um objeto Java, executando em uma *Máquina Virtual Java (JVM)*, podem ser invocado por objetos executados em outra *JVM*.

Aplicações em *RMI* envolvem dois aplicativos distintintos: cliente, responsável pela invocação de métodos remotos, e servidor, responsável pela criação de objetos remotos e referências para os mesmos. *Java RMI* assume que a rede de comunicação é uma coleção de *JVMs* homogêneas e que clientes e servidores são classes que executam nessas *JVMs*.

Ao contrário de *DCOM* e *CORBA*, *RMI* não utiliza uma linguagem declarativa do tipo *IDL* para definição de interfaces. Para tanto são usadas interfaces definidas na própria linguagem Java, o que resulta em outra diferença de *RMI* em relação as demais: a dependência de uma linguagem específica (Deitel e Deitel, 2001).

A característica mais importante de *RMI* é a possibilidade de passar objetos como parâmetros (passagem de parâmetro por valor) durante a invocação de um método (Gritzalis et al., 2000), sendo que para transmissão e organização dos parâmetros, *RMI* serializa os objetos antes de enviá-los pela rede de comunicação.

Esses objetos podem estar disponíveis (objeto local) ou não (objeto remoto) para o servidor responsável pela execução do método. Caso o objeto seja remoto, seu *bytecode* deve ser carregado no servidor, para que o método possa ser executado. Dessa forma, tanto cliente como servidor podem aumentar seus conjuntos básicos de tipos de dados.

O carregamento do *bytecode* implica uma pequena sobrecarga, decorrente dessa transmissão adicional. Contudo, apesar da sobrecarga, essa característica torna menos complexo o trabalho do desenvolvedor, uma vez que *IDLs* usadas em *DCOM* e *CORBA* possuem apenas tipos de dados primitivos.

Em qualquer aplicação de objetos distribuídos existe uma seqüência de passos genéricos que devem ser seguidos durante a requisição de um serviço remotamente executado. Esses passos são:

i. **Localizar o objeto remoto**: Esta etapa consiste em buscar a referência de um objeto remoto.

ii. **Comunicar-se com o objeto remoto**: Etapa transparente, uma vez que detalhes de comunicação são escondidos pela arquitetura e as chamadas a objetos remotos são idênticas às locais.

iii. **Carregar o *bytecode* de objetos passados por parâmetros**: Permite que o lado remoto carregue classes necessárias para execução do método.

A Figura 3.11 mostra um exemplo de aplicação *RMI* onde cliente e servidor utilizam uma terceira parte, *registro*, para registro e consulta de interfaces.

A Figura 3.11 demonstra as seguintes interações entre *Cliente* e *Servidor*:

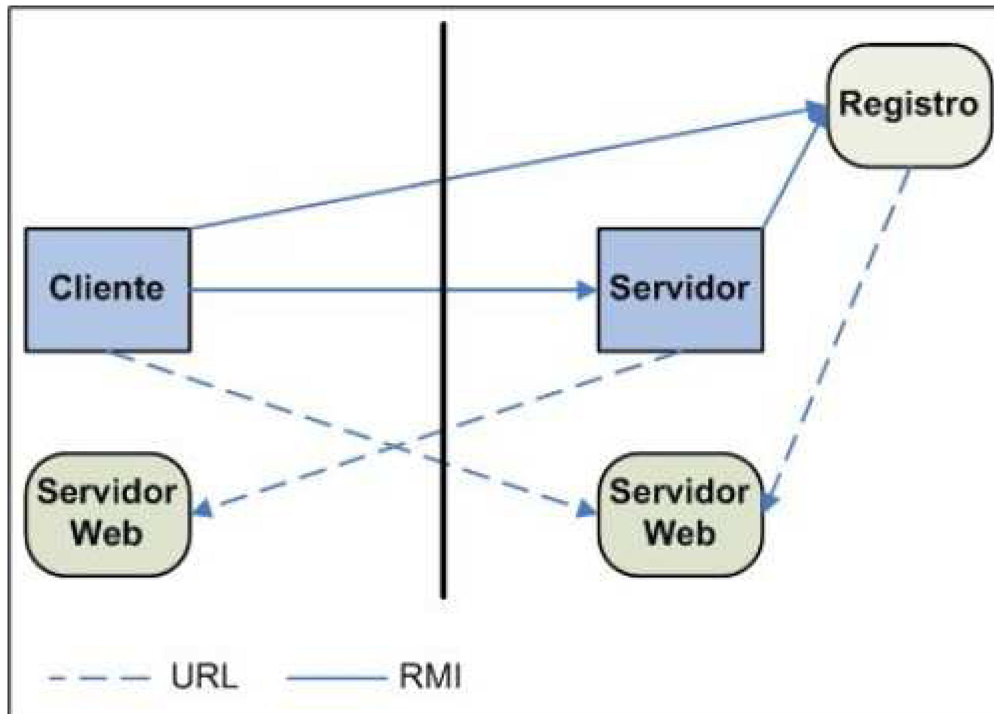


Figura 3.11: Exemplo de Aplicação RMI.

i. O *Servidor* cria os objetos e registra os mesmos no *Registro*.

ii. Para buscar uma referência a um objeto, o *Cliente* também acessa o *Registro*.

iii. Após receber a referência, o *Cliente* faz a requisição ao *Servidor*.

iv. O carregamento de *bytecodes* de objetos remotos são feitos utilizando o *Servidores Web* do lado do *Cliente* e do *Servidor*.

RMI também possui *Stubs* e *Skeletons* com a função de substitutos responsáveis pelas tarefas de *marshalling* e *unmarshalling* de parâmetros, além de permitir interação entre aplicações baseadas e não baseadas em Java, por meio de *Java IDL* (Jav, 2006).

3.5 Gerador de GUI com suporte a RT-RPC

O Gerador de *GUI* com suporte a *RT-RPC* (Villela, 2001) é uma ferramenta que auxilia o desenvolvedor na construção de aplicações com interfaces gráficas em *Java* que se

comunicam com aplicações em *C++* por meio de *RPCs* de tempo real. Essa ferramenta fornece uma *Interface Development Environment (IDE)* para desenvolvimento de aplicações clientes, classes e interfaces auxiliares, bem como dos módulos de comunicação que permitem as chamadas remotas de procedimento de tempo real.

Criado com o intuito de evitar a necessidade de aprendizado de novas tecnologias por parte do desenvolvedor, o Gerador de *GUI* com suporte a *RT-RPC* fornece um ambiente visual onde o desenvolvedor é capaz de gerar protótipos da *GUI* da aplicação sem preocupar-se com os detalhes de rede, abstraídos pelas *RPCs*.

O Gerador de *GUI* com suporte a *RT-RPC* é composto de 3 elementos, listados a seguir:

- **Editor de *GUI***: Permite a criação de aplicações visuais *Java*, utilizando o pacote *Abstract Window Tool (AWT)* fornecido com a *Interface de Programação de Aplicações (API)* da *Kit Java de Desenvolvimento (JDK)*. Esse editor ainda permite que sejam definidos métodos para manipulação de eventos de *GUI* (*e.g.* eventos de *mouse* e pressionamento de teclas).

- **Editor de código fonte**: Permite definir classes e interfaces auxiliares à aplicação, além de permitir a edição de código da aplicação visual.

- **Editor de procedimentos remotos**: Permite a definição de assinaturas e do corpo dos procedimentos remotos. Existem dois tipos de editores: editor de procedimentos remotos em *Java* e editor de procedimentos remotos em *C++*.

Depois do desenvolvimento da aplicação, é possível gerar arquivos de código fonte para os módulos de *Java* e de *C++*. Tais códigos devem ser compilados com compiladores fornecidos por outras partes, uma vez que a ferramenta não fornece compiladores para nenhuma dessas linguagens. A Figura 3.12 mostra os arquivos gerados pela ferramenta.

Pela Figura 3.12 observa-se que a ferramenta gera códigos tanto para a aplicação cliente quanto para aplicação servidora. Os módulos de comunicação, assim como os

Figura 3.12: Códigos Gerados pelo Gerador de GUI com Suporte a RT-RPC.

Fonte: Villela (2001)

procedimentos remotos, são gerados tanto em *Java* para o cliente, quanto em *C++* para o servidor. No entanto, a ferramenta gera apenas os módulos de comunicação utilizados pela aplicação servidora, não sendo responsável pela geração da mesma.

Os procedimentos remotos são definidos por sua assinatura que deve conter, entre outras informações, os tipos de dados de seus parâmetros. Os tipos de dados dos parâmetros devem ser padronizados para que cliente e servidor possam interagir e interpretar as informações que são passadas ao procedimento durante uma chamada. A ferramenta determina que somente tipos básicos de *Java* devem ser utilizados para parâmetros, quer estes sejam variáveis simples, quer sejam vetores. A Tabela 3.2 mostra os tipos permitidos na definição de procedimentos remotos.

Tabela 3.2: Tipos Básicos de Java.

Tipo	Nro. Bits	Descrição
long	64	Número inteiro com sinal de 64 bits
int	32	Número inteiro com sinal de 32 bits
short	16	Número inteiro com sinal de 16 bits
byte	8	Número inteiro com sinal de 8 bits
double	64	Número de ponto flutuante de 64 bits
float	32	Número de ponto flutuante de 32 bits
char	16	Caracter unicode
boolean	1	Valor booleano

As mensagens trocadas entre cliente e servidor têm um padrão definido para que ambos os módulos de comunicação possam interpretar as informações que estão sendo

transmitidas. São 3 as estruturas de mensagens definidas que fazem parte do protocolo de comunicação: estruturas de mensagens *RPC*, estruturas de mensagens *promise* e as estruturas de mensagens confirmação e cancelamento.

As *Promises* são tipos de dados criados para permitir chamadas remotas de procedimentos assíncronas, permitindo a um cliente, que executa uma chamada remota de procedimento, continuar com outras tarefas enquanto aguarda a execução do procedimento. Os objetos do tipo *Promise* possuem dois estados distintos: pronto e bloqueado. Todo objeto *Promise* recém criado entra em estado bloqueado até que uma mensagem *promise* de retorno de alguma chamada de procedimento preencha-o com algum valor. Já um objeto em estado pronto armazena os resultados do retorno de uma execução ou exceções que possam ter ocorrido durante a execução do procedimento. A estrutura de mensagens *promise* são necessárias para que o retorno da execução de um procedimento possa ser enviado a aplicação responsável pela chamada e possa ser encaminhado ao objeto *Promise* correto.

O Gerador de *GUI* com Suporte a *RT-RPC* não utiliza nenhum padrão de *RPC* daqueles apresentados neste capítulo. Ao invés disso, este cria um protocolo próprio de comunicação com formatos particulares de mensagens. A Figura 3.13 ilustra o formato escolhido para mensagens *RPC*.

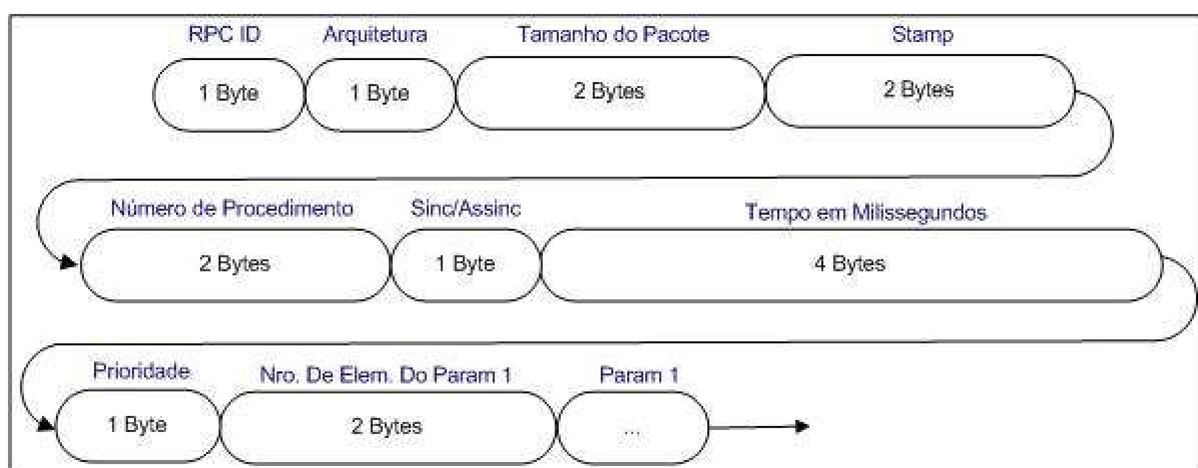


Figura 3.13: Estrutura de Mensagem RPC.

Pela Figura 3.13 observa-se que a mensagem é segmentada em várias partes. Cada uma das partes identifica uma característica dessa estrutura, as quais são descritas na Tabela 3.3.

Tabela 3.3: Descrição das Partes da Estrutura de Mensagem RPC.

Nome	**Nro. *Byte***	**Descrição**
RPC ID	1	Responsável por diferenciar esta estrutura das demais.
Arquitetura	2	Representa a arquitetura da máquina que executou a chamada ao procedimento.
Tamanho do Pacote	3 e 4	Representa o tamanho do pacote, permitindo a leitura do mesmo direto para um buffer.
Stamp	5 e 6	Permite direcionar mensagens *promise* para os respectivos objetos *Promise*.
Número do Procedimento	7 e 8	Identifica o procedimento que será executado. A partir da identificação é possível determinar tipos dos parâmetros.
Sínc/Assínc	9	Identifica o modo de execução, se síncrono ou assíncono.
Tempo em Milissegundos	10 a 13	Especifica o tempo máximo para execução de um procedimento.
Prioridade	14	Prioridade de execução do procedimento.
Nro. de Elem. do Param. 1	15 e 16	Especifica quantos elementos compõem o parâmetro que vem na seqüência.
Param 1	17 em diante	Valores do parâmetro descrito pelos bytes anteriores.

Após o primeiro parâmetro, diversos outros parâmetros podem ser concatenados, os quais devem ser descritos em pares da mesma forma que o primeiro: dois *bytes* indicando a quantidade de elementos e os valores seguindo o número de elementos.

As mensagens de *promise* são retornadas à aplicação cliente contendo o resultado da execução do procedimento remoto. Por mensagens desse tipo é possível saber se o procedimento cumpriu ou não a *deadline* especificada para sua execução. A Figura 3.14 ilustra a estrutura de mensagem *promise*.

A Figura 3.14 mostra que a estrutura de mensagens *promise* é bastante semelhante a estrutura de mensagens *RPC*, no entanto, mensagens *promise* apresentam campos para representação de modo de operação, tempo para execução e prioridade, campos estes que ocupam do nono ao décimo-quarto *byte* das mensagens *RPC*. Como as mensagens

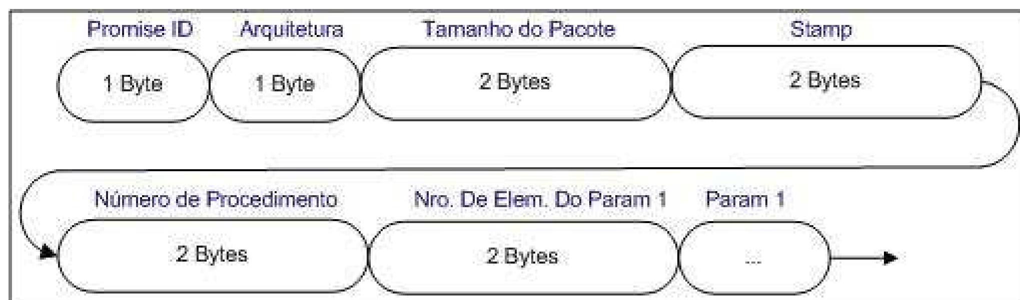


Figura 3.14: Estrutura de Mensagem Promise.

promises são utilizadas apenas para retorno, estas contêm apenas os campos necessários para encaminhamento da mensagem ao objeto *Promise* correto.

As mensagens de confirmação e cancelamento são enviadas das aplicação que fez a chamada a aplicação que executa a chamada e são necessárias para confirmar ou não o trabalho realizado pelo procedimento remoto. Enquanto as mensagens de confirmação são enviadas para procedimentos que cumpriram o tempo de execução, as mensagens de cancelamento são enviadas caso isso não ocorra. A Figura 3.15 ilustra a estrutura de mensagens de confirmação e cancelamento.

Figura 3.15: Estrutura de Mensagens de Confirmação e Cancelamento.

A estrutura de mensagens de confirmação ou cancelamento são menores que as demais, uma vez que apenas devem confirmar a validade da execução de um procedimento. A Tabela 3.4 apresenta a descrição de cada um dos campos mostrados na Figura 3.15.

O procedimento remoto só persiste a execução se receber uma mensagem de confirmação, caso contrário o estado anterior a execução é estabelecido.

O Gerador de *GUI* com suporte a *RT-RPC* é uma ferramenta que permite o desenvolvimento de aplicações que suportem chamadas remotas de procedimento de tempo real, no entanto, o desenvolvimento do cliente é bastante restrito uma vez que limita-se apenas ao

Tabela 3.4: Descrição das Partes da Estrutura de Mensagem de Confirmação ou Cancelamento.

Nome	**Nro. *Byte***	**Descrição**
ConfCanc ID	1	Responsável por diferenciar esta estrutura das demais.
Arquitetura	2	Representa a arquitetura da máquina que executou a chamada ao procedimento.
Conf/Canc	3	Possui o valor 0 para mensagem de confirmação e valor 0 para mensagem de cancelamento.
stamp	5 e 6	Permite direcionar mensagens para o procedimento remoto de mesmo *stamp*.

desenvolvimento em linguagem *Java*, obrigando o usuário a possuir uma *JVM* instalada na máquina hospedeira. As estruturas de mensagens são bem definidas e permitem a interoperabilidade com outras linguagens, porém, tais características não são implementadas na versão atual da ferramenta. A ferramenta também desconsidera as características do dispositivo hospedeiro da aplicação cliente e da aplicação servidora, deixando em aberto situações em que o usuário possui dispositivos com recursos limitados para executar as aplicações geradas.

Capítulo 4

Fundamentação Matemática

Recorrer a matemática tem por objetivo buscar soluções precisas e confiáveis. Este capítulo descreve duas áreas de conhecimento matemáticas que auxiliarão o desenvolvimento deste projeto.

4.1 Teoria de Filas

Segundo Larson e Odoni (1981) a Teoria de Filas é a área da Pesquisa Operacional que explora as relações entre a demanda por serviços em um sistema e os tempos de esperas pelos usuários de tal sistema. Nessa definição é possível identificar os dois aspectos que caracterizam um sistema de filas:

i. processador, que fornece serviços;

ii. os consumidores de tal serviço, que, muitas vezes, devem esperar para recebê-lo.

No início da década de 50, *Kendall* apresentou o conceito de modelo básico de filas e uma notação que permitia a identificação de alguns elementos desse modelo (Kendall, 1951). Os desenvolvimentos em Pesquisa Operacional contribuíram de maneira abrangente

para o desenvolvimento dos conceitos e modelagem em Teoria de Filas, aspectos que até hoje são alvo de publicações em importantes periódicos internacionais.

4.1.1 Filas

As filas existem devido a necessidade de compartilhamento de recursos escassos. Um modelo básico de sistema de filas considera a demanda por um serviço e o fornecimento do mesmo, como pode ser visto na Figura 4.1.

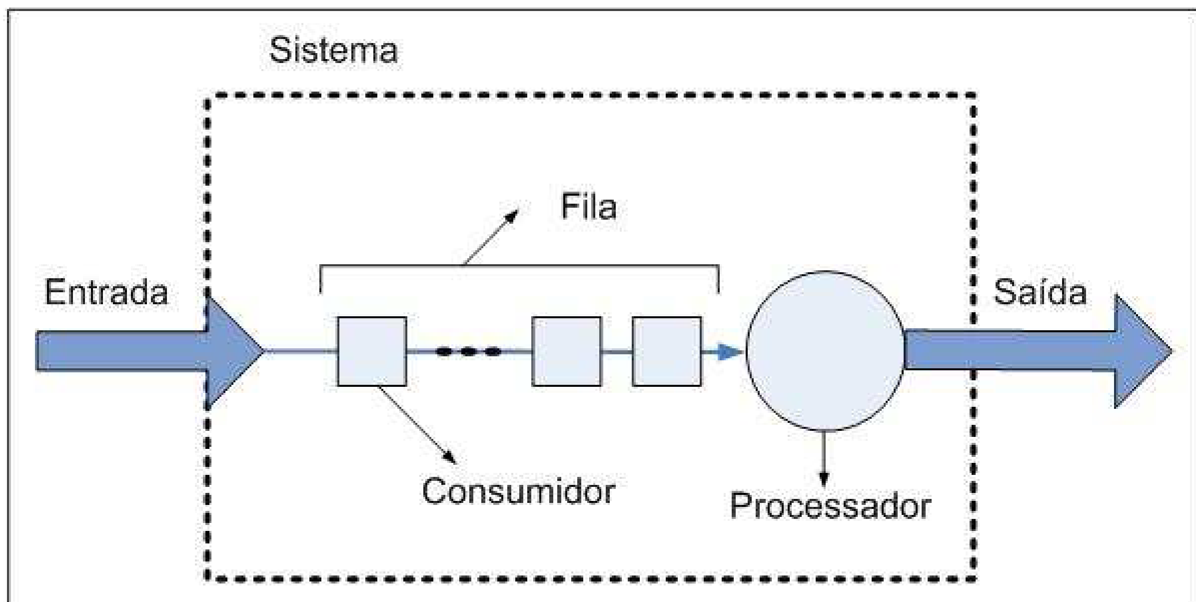


Figura 4.1: Modelo de Fila.

A Figura 4.1 apresenta os seguintes componentes:

- **Entrada**: entrada de consumidores para receberem algum tipo de serviço no sistema, ou seja, é a chegada de consumidores no sistema. Os consumidores chegam em intervalos de tempo que são independentes e podem ter uma determinada distribuição de probabilidade comum.
- **Saída**: saída de consumidores após o processamento;
- **Consumidores**: podem ser clientes, pacotes, processos, tarefas que estão aguardando o processamento;

- **Processador**: estágio responsável pelo processamento, ou seja, o fornecimento do serviço que os consumidores demandam. Os tempos de processamento são considerados independentes e com alguma distribuição probabilística em comum;

- **Fila**: quantidade de consumidores que está aguardando processamento.

Outras características que podem ser consideradas em um sistema de filas são:

- Comportamento dos consumidores: tolerância à espera, ou seja, o quanto os consumidores estão dispostos a esperar pelo serviço e como o tamanho da fila influência na decisão do consumidor a respeito de sua entrada no sistema;

- Capacidade da fila: quantidade máxima de consumidores que pode ficar aguardando por processamento;

- Capacidade do sistema: abrange o tamanho da fila e o número de consumidores sendo atendidos;

- Sala de espera: uma espécie de anexo ao sistema, onde os consumidores aguardam para entrar no sistema;

- Disciplina da fila: determinação das políticas de atendimento dos consumidores.

A notação de *Kendall* reúne algumas dessas características da seguinte forma:

A / S / m / B / K / SD

Cada um desses itens podem ser entendidos como:

- **A**: processo de chegada dos consumidores, que pode ser M (Exponencial Negativa, ou *Poisson*), Ek (*Erlang* com k-ésima ordem), Hk (Hiper-Exponencial com k-ésima ordem), D (Determinística) ou G (Geral, isto é, qualquer distribuição).

- **S**: distribuição do tempo de processamento, que pode ser M, Ek, Hk, D ou G.

- **m**: número de processadores.
- **B**: capacidade do sistema (omitida quanto é infinita).
- **K**: tamanho da população (omitida quando infinita).
- **SD**: disciplina de serviço (omitida quando o primeiro consumidor a chegar no sistema é o primeiro a ser atendido).

Dessa forma, um sistema M / M / 1, é um sistema que apresenta processo de chegada exponencial, distribuição exponencial do tempo de processamento, um único processador, a capacidade do sistema e a população são infinitas e a disciplina de atendimento da fila é por ordem de chegada.

A Disciplina das Filas diz, portanto, à respeito da ordem em que os consumidores serão atendidos e segue os critérios estabelecidos pelo sistema, como:

- ***First Come, First Served (FCFS)***: a ordem de atendimento obedece a ordem de chegada, ou seja, o primeiro consumidor a entrar no sistema será o primeiro a ser atendido.
- ***First Come, Last Served (FCLS)***: nesta disciplina o ultimo consumidor a entrar no sistema será o primeiro a ser atendido.
- ***Service In Random Order (SIRO)***: os consumidores são atendidos em ordem aleatória.
- **Round Robin**: o processador escolhe um consumidor da fila e permite seu processamento durante um *quantum* de tempo pré-estabelecido; se a necessidade de processamento do consumidor for menor do que o *quantum*, assim que o processamento é finalizado, o processador é liberado para um próximo consumidor, caso contrário, o processamento é interrompido e um novo consumidor é escolhido.

- ***Menor Tempo de Processamento (SPT)***: a fila é ordenada de forma que o primeiro consumidor a ser atendido seja o com a necessidade de processamento menor.

- ***Shortest Expected Processing Time (SEPT)***: a fila é ordenada de forma que o primeiro consumidor a ser atendido seja o com a necessidade esperada de processamento menor.

- **Prioridades**: critérios pré-estabelecidos classificam e, assim, indicam a ordem de atendimento dos consumidores.

- **Preempção**: possibilidade de interrupção do processamento que pode ser ”resumível”, ou seja, a posterior continuação do processamento se inicia no estágio em que foi interrompido ou ”não-resumível”, isto é, o processamento realizado até o momento da interrupção é descartado.

4.1.2 Tipos de Filas

Segundo Maister (1995) a primeira diferenciação de sistemas de filas é a quantidade de estágios de processamento, assim, esses sistemas podem ser classificados como: sistema único estágio ou sistema múltiplo estágio.

Sistema Único Estágio: o sistema apresenta apenas um estágio de processamento.

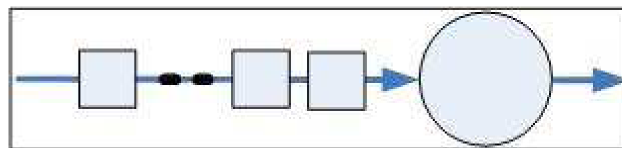


Figura 4.2: Sistema Único Estágio.

Sistema Múltiplo Estágio: os consumidores devem passar por diversos estágios de processamento antes de deixarem o sistema. A Figura 4.3 ilustra esse sistema.

A quantidade de estágios de um sistema de filas permite a análise de quantas filas um consumidor de um serviço deve passar até sair do sistema, assim, pode-se, em conjunto

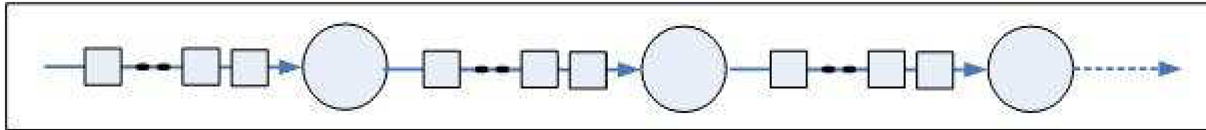


Figura 4.3: Sistema Múltiplo Estágio.

com os tempos esperados de processamento, determinar a tempo esperado de permanência do consumidor no sistema.

Outras formas de classificação dos tipos de fila, segundo Maister (1995) e, considerando o número de processadores e a quantidade de filas, são:

Sistema Único Estágio Paralelo: Vários processadores estão dispostos em paralelo, cada um, precedido por uma fila. Assim, o consumidor ao entrar no sistema escolhe, ou é direcionado, a um processador, como a escolha de caixas de pagamento em um supermercado, por exemplo.

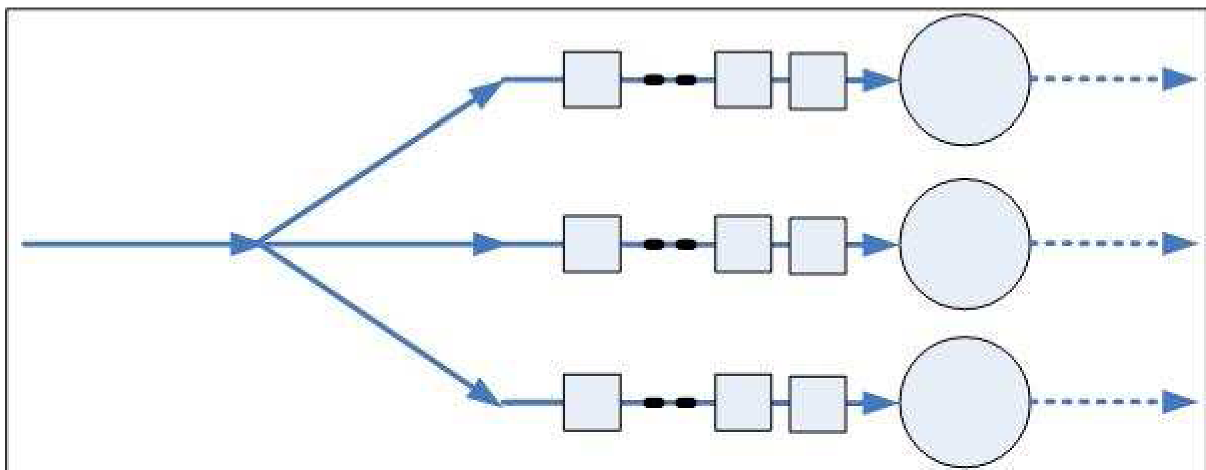


Figura 4.4: Sistema Único Estágio Paralelo.

Sistema Multicanal Único Estágio: Vários processadores são dispostos em paralelo, porém precedidos por uma fila única, como o caso de fila única para atendimento a clientes em bancos.

Sistemas Multi-filas: Um único processador é precedido por mais de uma fila, como em um semáforo em um cruzamento de vias.

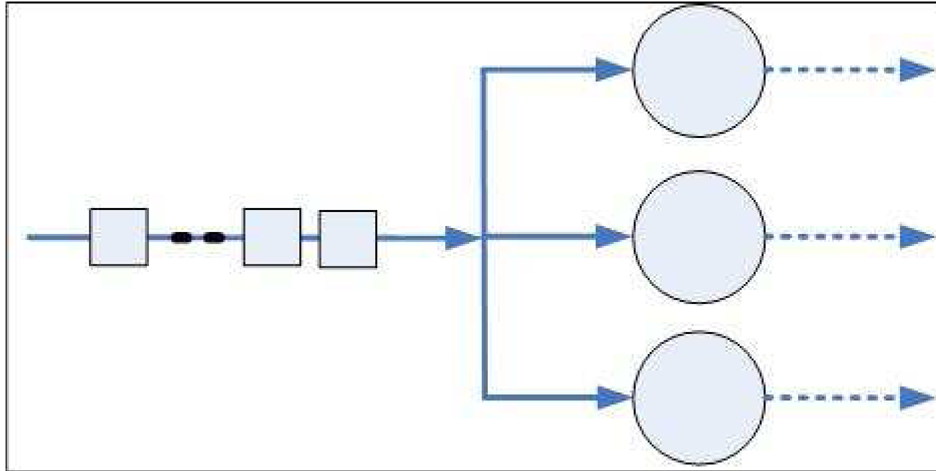


Figura 4.5: Sistema Multicanal Único Estágio.

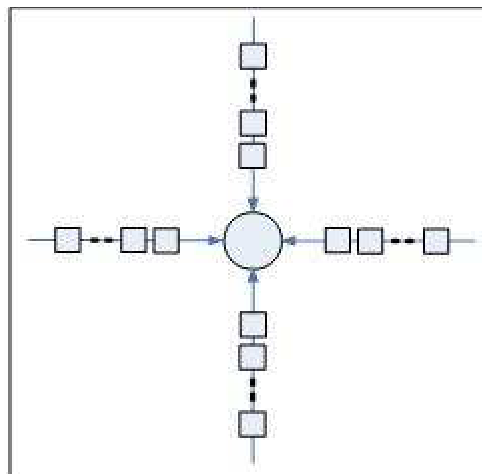


Figura 4.6: Sistema Multifilas.

Sistema Discriminatório de Clientes: Os clientes são segregados antes de serem direcionados às filas dos processadores, assim, cada processador se especializa no atendimento de um determinado tipo de consumidor, como em operações de *check-in* de aeroportos.

4.1.3 Medidas de Interesse

Um sistema e filas pode ser analisado de diversas maneiras e para tanto, algumas notações devem ser previamente definidas. Considerando a Figura 4.8, podem-se identificar algumas medidas de interesse.

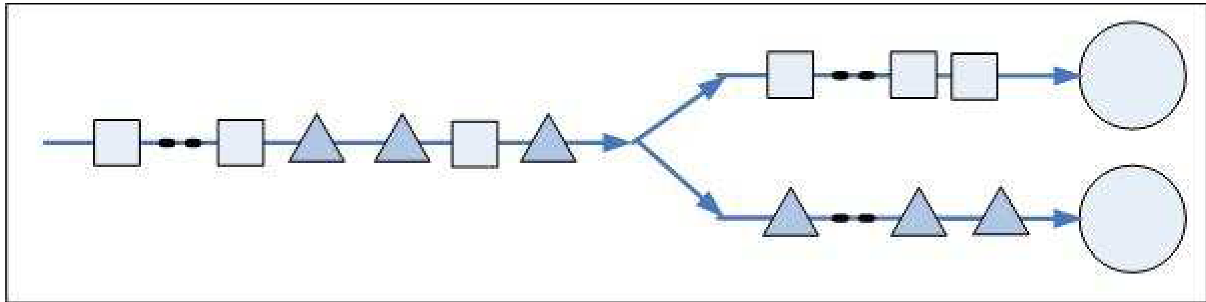


Figura 4.7: Sistema Discriminatório de Clientes.

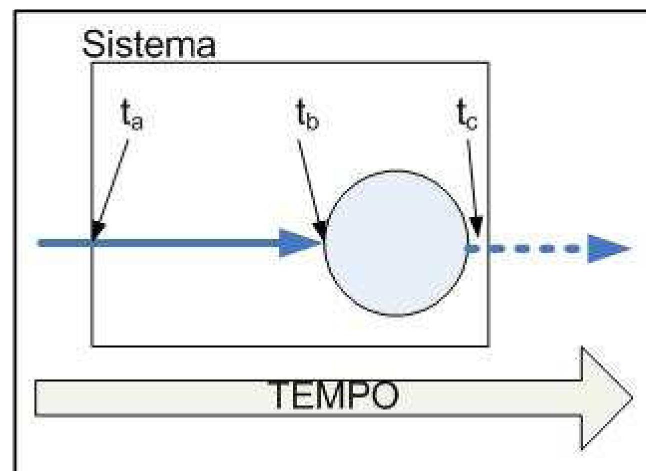


Figura 4.8: Eventos Sistema de Filas.

Os instantes ta, tb e tc representam eventos para o i-ésimo consumidor, de chegada no sistema, de início de processamento e de saída do sistema, respectivamente. A terminologia apresentada a seguir segue o padrão e as definições estabelecidos por *Larson* e *Odoni* apresentados em seu livro (Larson e Odoni, 1981) e representam as medidas para a análise do comportamento de um sistema de filas.

Pela Figura 4.8 pode-se definir:

- O intervalo de tempo entre chegadas do i-ésimo consumidor e seu antecessor ao sistema, mostrado pela função 4.1.

$$X(i) \equiv t_a(i) - t_a(i-1) \tag{4.1}$$

- O tempo de processamento para i-ésimo consumidor, mostrado pela função 4.2.

$$S(i) \equiv t_c(i) - t_b(i) \tag{4.2}$$

Outros valores podem ser obtidos a partir das funções 4.1 e 4.2. Considerando $E[\,]$ como o valor esperado ou esperança de uma variável, tem-se que:

- O tempo esperado, ou esperança, de processamento, mostrado pela função 4.3.

$$E[S] \equiv 1/\mu \tag{4.3}$$

  Sendo a função 4.3 a esperança de um processamento, pode-se concluir que μ é a taxa de serviço, ou seja, o número de consumidores processados em uma unidade de tempo.

- O tempo esperado, ou esperança, entre as chegadas ao sistemas, mostrado pela função 4.4.

$$E[X] \equiv 1/\lambda \tag{4.4}$$

  Sendo a função 4.4 a esperança entre as chegadas, pode-se concluir que λ é o número de chegadas ao sistema por unidade de tempo.

Relacionando μ e λ pode ser obtida a medida ρ, que representa a intensidade de fluxo e é mostrada pela equação 4.5.

$$\rho = \lambda/\mu = \lambda E[S] \tag{4.5}$$

Em um sistema com mais de um processador (m), pode-se obter, a partir de ρ, a taxa de utilização do sistema denotada por U e mostrada pela equação 4.6.

$$U = \rho/m = \lambda/m\mu = \lambda\, E[S]/m \tag{4.6}$$

Ainda pela Figura 4.8 é possível definir seguintes funções:

- Tempo de espera em fila do i-ésimo consumidor, mostrado pela função 4.7.

$$W_q(i) \equiv t_b(i) - t_a(i) \tag{4.7}$$

- Tempo total de permanência do i-ésimo consumidor no sistema, mostrado pela função 4.8.

$$W(i) \equiv t_c(i) - t_a(i) \tag{4.8}$$

Dessa forma, outra maneira de definir $W(i)$ é dada pela função 4.9:

$$W(i) = W_q(i) + S(i) \tag{4.9}$$

Na condição de equilíbrio do sistema, outras duas medidas de interesse podem ser:

- O tempo esperado de ocupação de um sistema por um consumidor, representado pela função 4.10.

$$\overline{W}(i) \equiv E[W] = \lim_{i \to \infty} E[W(i)] \tag{4.10}$$

- O tempo esperado por um consumidor na fila, representado pela função 4.11.

$$\overline{W_q}(i) \equiv E[W_q] = \lim_{i \to \infty} E[W_q(i)] \tag{4.11}$$

Considerando um momento aleatório em um sistema em funcionamento, $N(t)$ equivale ao número total de consumidores no sistema (soma dos consumidores em fila e os consumidores sendo processados) no instante t, enquanto que $N_q(t)$ equivale ao número de consumidores na fila, no instante t. Assim, as seguintes medidas podem ser definidas:

- número total esperado de consumidores no sistema, representado pela equação 4.12.

$$\overline{L} \equiv E[N] = \lim_{t \to \infty} E[N(t)] \tag{4.12}$$

- número de esperados de consumidores na fila, representado pela equação 4.13.

$$\overline{L_q} \equiv E[N_q] = \lim_{t \to \infty} E[N_q(t)] \tag{4.13}$$

Um importante resultado definido em Teoria de Filas é a Lei de Little (Little, 1961), que apresenta a relação entre o número de consumidores no sistema (L), o tempo de permanência do consumidor no sistema (W), e a taxa de chegada (λ), como mostrado pela equação 4.14.

$$L = \lambda W \tag{4.14}$$

Considerando um sistema, que no instante $t = 0$ está vazio, a acumulação do número de chegadas e saídas, em um intervalo aleatório τ, pode ser vista na Figura 4.9.

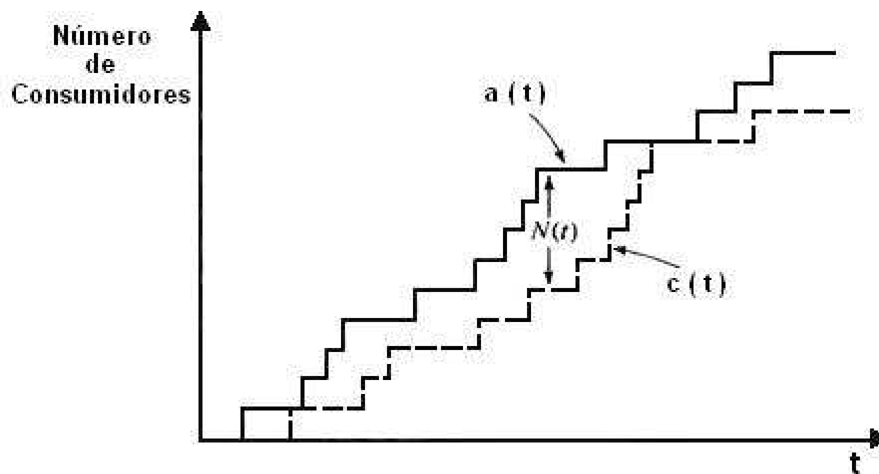


Figura 4.9: Distribuição Acumulada de Clientes no Sistema.

Fonte: Larson e Odoni (1981)

Na Figura 4.9, $a(\tau)$ é definido como o número de chegadas ao sistema no intervalo $[0, \tau]$, enquanto $c(\tau)$ é definido como o número de processamentos no intervalo $[0, \tau]$. Dessa forma o número de consumidores no instante $t = \tau$ é dado pela função 4.15.

$$N(\tau) = a(\tau) - c(\tau) \tag{4.15}$$

Pelas funções $a(\tau)$ e $c(\tau)$ é possível identificar o tempo gasto pelos consumidores no sistema, no intervalo $[0, \tau]$, como mostrado pela função 4.16:

$$l(\tau) = \int_0^T [a(t) - c(t)]dt \tag{4.16}$$

Sendo assim, o número médio de consumidores ($\hat{N}$) no intervalo τ pode ser determinado pela relação do total de tempo gasto pelos consumidores e o tempo τ, mostrado pela função 4.17:

$$\hat{N}(\tau) = l(\tau)/\tau \tag{4.17}$$

Multiplicando a direita da igualdade por $a(\tau)/a(\tau)$, tem-se:

$$\hat{N}(\tau) = \frac{l(\tau)}{a(\tau)} \frac{a(\tau)}{\tau}$$

Como $a(\tau)/\tau$ é a média de chegadas durante o intervalo τ, chega-se a:

$$\frac{a(\tau)}{\tau} = \hat{\lambda_\tau}$$

De maneira similar, $l(\tau)/\tau$ é a média de tempo gasto pelos consumidores no sistema, e, portanto:

$$\frac{l(\tau)}{\tau} = \hat{W}_\tau$$

Sendo assim, pode-se obter o seguinte resultado mostrado pela função 4.18:

$$\hat{N}(\tau) = \hat{\lambda}_\tau \hat{W}_\tau \tag{4.18}$$

Se o intervalo de tempo $\tau \to \infty$, $\overline{L}$, λ, $\overline{W}$ representam os limites de $\hat{N}(\tau)$, $\hat{\lambda}_\tau$ e $\hat{W}_\tau$ e, portanto:

$$\overline{L} = \lambda \overline{W}$$

Similarmente, se apenas os momentos de entrada e saída da fila forem considerados, tem-se:

$$\overline{L_q} = \lambda \overline{W_q}$$

Se o sistema (estrutura) for o foco da análise, pode-se obter:

$$\overline{L_s} = \lambda E[S] = \lambda / \mu$$

e, $\overline{L_s}$ é o número médio de pessoas no sistema em equilíbrio. A última relação que é possível definir é mostrada pela equação 4.19:

$$\overline{W} = E[S] + \overline{W_q} = \frac{1}{\mu} + \overline{W_q} \tag{4.19}$$

Assim, $\overline{L}$, $\overline{L_q}$, $\overline{W}$ e $\overline{W_q}$ estão relacionados e se uma dessas medidas é conhecida, bem como as taxas λ e μ, as demais medidas podem ser determinadas.

Essas medidas são importantes para se determinar o comportamento do sistema e determinar maneiras de melhor gerenciar o mesmo. Por exemplo, sabendo-se que o número médio de consumidores em fila é alto, pode-se determinar uma duplicação do número de processadores na tentativa de diminuir o tempo de espera.

Em seu trabalho Maister (1995) apresenta um exemplo que ilustra como essas medidas podem ser utilizadas para se analisar o comportamento do sistema:

Supondo um sistema com um único processador, que pode servir 70 clientes por hora em média (isto implica em um tempo de serviço médio de 60/70 = 0,86 minuto por cliente). Clientes chegam em uma taxa média de 49 por hora. Assim, tem-se:

m = 1 servidor

μ = 70 pessoas por hora

λ = 49 pessoas por hora

ρ = 49 / 70 = 0,7

$W_q = L_q$ / = 1,633 / 49 = 0,033 horas ou 2 minutos.

$L = L_q$ + = 1,633 + 0,7 = 2,333 pessoas.

$W = W_q$ + 1 / = 0,033 + 1 / 70 horas = 2,86 minutos (2 minutos de tempo médio em fila e 0,86 minutos de tempo médio de serviço).

$U = \lambda/M\mu$ = 49 / 1.70 = 0,7 (em 30 % do tempo o servidor estará ocioso).

A Tabela 4.1 sumariza esses resultados.

Analisando esses resultados, algumas questões podem ser feitas:

λ	μ	m	ρ	L_q	W_q	L	W	U
49	70	1	0,7	1,63	2	2,3	2,86	0,7

Tabela 4.1: Tabela de Dados do Exemplo.

Algumas questões podem ser feitas:

i. O que acontece se o número de servidores for duplicado, mantendo o sistema FCFS e uma fila para dois servidores?

   m = 2 servidores

   $\rho = 49/70 = 0,7$

As medidas de desempenho estão apresentadas na Tabela 4.2. Uma análise dessas medidas permite a verificação de que, dobrar o número de servidores resulta em dividir mais que pela metade o comprimento da fila.

λ	μ	m	ρ	L_q	W_q	L	W	U
49	70	1	0,7	1,63	2	2,3	2,86	0,7
49	**70**	**2**	**0,7**	**0,098**	**0,12**	**0,8**	**0,98**	**0,35**

Tabela 4.2: Tabela de Desempenho da Primeira Questão.

ii. O que acontece se o tempo de serviço médio for cortado pela metade ao invés de se duplicar o número de servidores?

Cortando o tempo de serviço médio pela metade, dobra-se o número de clientes que podem ser servidos por hora. Então:

m = 1 servidor

μ = 140 pessoas por hora

ρ = 49 / 70 = 0,35

As medidas de desempenho, para esse caso, estão apresentadas na Tabela 4.3.

λ	μ	m	ρ	L_q	W_q	L	W	U
49	70	1	0,7	1,63	2	2,3	2,86	0,7
49	70	2	0,7	0,098	0,12	0,8	0,98	0,35
49	**140**	**1**	**0,35**	**0,188**	**0,23**	**0,54**	**0,66**	**0,35**

Tabela 4.3: Tabela de Desempenho da Segunda Questão.

Pode se observar, pela Tabela 4.3 que o tempo de permanência no sistema (W) diminui.

iii. O que acontece com a fila se um segundo sistema é aberto tal que metade dos clientes passam a freqüentar esse novo sistema e a outra metade permanece no antigo, apresentando uma situação de duas filas independentes?

Nessa situação, a taxa de chegada média será metade da original e os valores se aplicarão a cada uma das duas lojas:

m = 1 servidor

μ = 70 pessoas por hora

λ = 24,5 pessoas por hora

ρ = 24,5 / 70 = 0,35

As medidas de desempenho estão apresentadas na Tabela 4.4.

λ	μ	m	ρ	L_q	W_q	L	W	U
49	70	1	0,7	1,63	2	2,3	2,86	0,7
49	70	2	0,7	0,098	0,12	0,8	0,98	0,35
49	140	1	0,35	0,188	0,23	0,54	0,66	0,35
24,5	**70**	**1**	**0,35**	**0,188**	**0,46**	**0,54**	**1,32**	**0,35**

Tabela 4.4: Tabela de Desempenho da Terceira Questão.

Comparando-se a linha 2 (dois servidores e uma fila de espera), da Tabela 4.4, com a linha 4 (dois servidores e 2 filas independentes) é possível observar uma diferença de 65% entre o tempo de permanência no sistema. A disciplina da fila influencia nessa medida, pois, na linha 2, a existência de uma única fila faz com que os clientes sejam atendidos em ordem de chegada, pelo primeiro servidor a ficar desocupado. Já no caso de duas filas independentes, o cliente é forçado a escolher uma das duas filas.

Esse exemplo ilustra como as medidas de desempenho podem ser utilizadas para se avaliar a melhor forma de se organizar um sistema de filas. Assim, tais análises podem ser consideradas de importância estratégica, uma vez que possibilitam o projeto de um sistema que priorize o atendimento ao cliente, melhorando-o e, conseqüentemente contribuindo para a sobrevivência de tal sistema à longo prazo.

4.2 Séries Temporais

Segundo Morettin e Toloi (1981b), séries temporais são conjuntos de observações ordenadas ao longo de intervalos temporais, e são compostas por quatro elementos distintos:

i. **Tendência:** que indica o sentido de deslocamento das observações;

ii. **Ciclo:** que é um movimento ondulatório periódico que se repete em vários intervalos de tempo;

iii. **Sazonalidade:**, que é um movimento ondulatório de curta duração; e

iv. **Ruído aleatório:**, que é a parte dos dados de variabilidade intrínseca e que não pode ser modelada.

Sendo um processo estocástico um conjunto de todas as possíveis observações de um evento, então cada possível trajetória observada é uma série temporal (Morettin e Toloi, 1981a). Existem diferentes interesses pelos quais estudar séries temporais. Que podem ser:

i. Investigar o mecanismo gerador da série.

ii. Fazer previsões de valores futuros.

iii. Descrever apenas seu comportamento.

iv. Procurar periodicidades relevantes nos dados.

Para todos os casos são construídos modelos probabilísticos que devem manter a simplicidade e serem os mais parcimoniosos possíveis, ou seja, apresentarem apenas os parâmetros realmente indispensáveis. Neste trabalho, o objetivo principal no uso de séries temporais foi previsão de valores que uma série poderia assumir.

Sempre é possível considerar um número muito grande de modelos diferentes para descrever o comportamento de uma série em particular, alguns modelos têm melhor ajuste, enquanto outros não se ajustam muito bem a série. A construção de um modelo depende, entre outras coisas, do conhecimento prévio da natureza da série e da existência de métodos de estimação suficientes.

Dentro dos modelos de séries temporais paramétricos, ou seja, aqueles que possuem um número finito de variáveis, um método é conhecido por ter modelos que se ajustam bem às séries estacionárias e não estacionárias, e às séries que possuem ou não dependência entre os dados. Este método reúne os modelos de *Box-Jenkins* (Box e Jenkins, 1976) que, ao contrário dos métodos baseados em suavização exponencial, exploram a possível dependência entre os dados observados. O método *Box-Jenkins* estabelece modelos para conjuntos de dados que apresentam comportamento estacionário, não estacionário ou sazonal. Estes modelos podem ser combinados para abranger o maior número possível de séries temporais, e são explorados com detalhes na próxima seção.

4.2.1 Método Box-Jenkins

Os modelos de *Box-Jenkins* são conhecidos, dentre os modelos paramétricos, por ajustarem-se bem às séries estacionárias e não estacionárias, e às séries que possuem ou não dependência entre os dados. O método consiste do ajuste de modelos auto-regressivos, integrados e de médias móveis. O método *Box-Jenkins* envolve modelos estacionários, não estacionários e sazonais.

4.2.1.1 Modelos Estacionários

Os modelos estacionários são os modelos que assumem que o processo está em equilibrado. Um processo estocástico pode ser classificado de 2 (duas) formas com relação a estacionaridade:

- Processo fracamente estacionário

- Processo fortemente estacionário

Um processo é considerado fracamente estacionário se a média e variância se mantêm constantes ao longo do tempo e a função de autocovariância depende apenas da defasagem entre os instantes de tempo. Um processo é fortemente estacionário se todos os momentos conjuntos são invariantes a translações no tempo.

O primeiro dos modelos de *Box-Jenkins* para processos estacionários é o modelo *Auto-regressivo (AR)* ou auto-regressivo.

Modelo Auto-Regressivo

No modelo auto-regressivo uma série temporal Z_t é descrita por seus valores anteriores regredidos e pelo ruído aleatório ϵ_t. Dessa forma, o modelo auto-regressivo com p parâmetros, ou seja, um *AR(p)*, pode ser dado pela Equação 4.20.

$$\tilde{Z}_t = \phi_1 \tilde{Z}_{t-1} + \phi_2 \tilde{Z}_{t-2} + \ldots + \phi_p \tilde{Z}_{t-p} + \epsilon_t \tag{4.20}$$

Segundo Johnson e Montgomery (1974) é conveniente trabalhar com séries temporais definidas em termos de desvios em relação a média μ da série. Sendo assim, na equação 4.20, temos que $\tilde{Z}_t = Z_t - \mu$, onde Z_t é o valor da série propriamente dito.

O relacionamento entre Z_t e seus valores anteriores Z_{t-i} para $i = 0, 1, 2, \ldots, p$ é dado pelos parâmetros ϕ_i. A equação de um modelo pode ser reescrita utilizando o operador de defasagem ou translação para passado, B. Este operador é definido pela Equação 4.21.

$$BZ_t = Z_{t-1} \tag{4.21}$$

A equação 4.20 para o modelo *AR(p)* pode ser reescrita com o operador de translado para o passado, resultando na Equação 4.22.

$$(1 - \phi_1 B - \phi_2 B^2 - \ldots - \phi_p B^p)\tilde{Z}_t = \Phi(B)\tilde{Z}_t = \epsilon_t \tag{4.22}$$

Os modelos auto-regressivos mais utilizados são aqueles de ordem 1 (um) e 2 (dois). O modelo *AR(1)* é a versão mais simples dos modelos auto-regressivos. As Equações 4.23 e 4.24 definem estes modelos.

$$\tilde{Z}_t = \phi_1 \tilde{Z}_{t-1} + \epsilon_t \tag{4.23}$$

$$\tilde{Z}_t = \phi_1 \tilde{Z}_{t-1} + \phi_2 \tilde{Z}_{t-2} + \epsilon_t \tag{4.24}$$

Segundo Werner e Ribeiro (2003), para o modelo ser estacionário é necessário que $|\phi_1| < 1$ (condição de estacionaridade) e que as autocovariâncias γ_k sejam independentes. No caso do modelo *AR(1)*, as autocovariâncias são dadas pela Equação 4.25, enquanto as autocorrelações ρ_k são dadas pela equação 4.26.

$$\gamma_k = \phi_1^k \gamma_0 \tag{4.25}$$

$$\rho_k = \frac{\gamma_k}{\gamma_0} = \phi_1^k \quad k = 0, 1, 2, \ldots \tag{4.26}$$

Quando ϕ_1 é positivo, a *Função de Autocorrelação (ACF)* ou função de autocorrelação ou decai exponencialmente, o mesmo acontece quando ϕ_1 é negativo, no entanto, os valores da função alternam entre positivos e negativos.

Modelos de Médias Móveis

Nos modelos *Média Móvel (MA)* ou médias móveis, a série Z_t é obtida da combinação dos ruídos aleatórios ϵ do período atual e dos períodos anteriores. A Equação 4.27 define o modelo *MA* de ordem q, ou simplesmente, *MA(q)*.

$$\tilde{Z}_t = \epsilon_t + \theta_1 \epsilon t - 1 + \theta_2 \epsilon t - 2 + \ldots + \theta_q \epsilon t - q \tag{4.27}$$

Da mesma forma que para os modelos auto-regressivos, $\tilde{Z}_t$ é definido como o desvio com relação a média da série, ou seja, $\tilde{Z}_t = Z_t - \mu$.

Os modelos auto-regressivos também podem ser escritos alternativamente utilizando-se o operador de translação para o passado B. A Equação 4.28 define o modelo MAq reescrito com o operador B.

$$(1 - \theta_1 B - \theta_2 B^2 - \ldots - \theta_q B^q)\epsilon_t = \Theta(B)\epsilon_t = \tilde{Z}_t \tag{4.28}$$

A versão mais simples e mais utilizada do modelo de médias móveis é o *MA(1)*, definido pela Equação 4.29.

$$\tilde{Z}_t = \epsilon_1 + \theta_1 \epsilon_{t-1} \tag{4.29}$$

A autocorrelação ρ é dada pela Equação 4.30.

$$\rho_1 = \frac{\gamma_1}{\gamma_0} = \frac{-\theta_1 \sigma_\epsilon^2}{(1+\theta_1^2)\sigma_\epsilon^2} = \frac{-\theta_1}{(1+\theta_1^2)} \qquad \rho_k = 0 \qquad k > 1 \tag{4.30}$$

Modelo Auto-Regressivo de Média Móvel

O modelo *Auto-Regressivo Média Móvel (ARMA)* ou auto-regressivo de média móvel é uma combinação dos modelos *AR* e *MA*. Alguns casos de modelos *AR* ou *MA* podem exigir um grande número de parâmetros. A mistura de filtros *AR* e *MA* pode gerar um modelo mais parcimonioso e que alcance os mesmos resultados. A Equação 4.31 mostra o modelo *ARMA* com parâmetros p para o filtro *AR* e q para o filtro *MA*, ou simplesmente, *ARMA(p,q)*.

$$\tilde{Z}_t = \phi_1 \tilde{Z}_{t-1} + \ldots + \phi_p \tilde{Z}_{t-p} + \epsilon_t - \theta_1 \epsilon_{t-1} - \ldots - \theta_q \epsilon_{t-q} \tag{4.31}$$

O representante mais simples do modelo *ARMA* é o *ARMA(1,1)* como mostrado pela Equação 4.32, enquanto sua função de autocorrelação é dada pela Equação 4.33.

$$\tilde{Z}_t = \phi_1 \tilde{Z}_{t-1} + \epsilon_t - \theta_1 \epsilon_{t-1} \tag{4.32}$$

$$\rho_1 = \frac{(1-\phi_1\theta_1)(\phi_1-\theta_1)}{1+\theta_1^2+2\phi_1\theta_1} \qquad \rho_k = \phi_1 \rho_{k-1} \quad para\ k > 1 \tag{4.33}$$

É possível verificar que os modelos *ARMA(1,0)* e *ARMA(0,1)* são idênticos aos modelos *AR(1)* e *MA(1)*, respectivamente, ou seja, o valor 0 (zero) como parâmetro em p ou q anula um dos componentes do modelo, e por isso o modelo *ARMA* consegue cobrir todos os casos de séries estacionárias.

4.2.1.2 Modelo Não Estacionário

Quando uma série temporal apresenta média e variância dependentes do tempo, é porque ela não é estacionária. A não-estacionariedade de uma série implica que: a) há inclinação nos dados e eles não permanecem ao redor de uma linha horizontal ao longo do tempo e/ou b) a variação dos dados não permanece essencialmente constante sobre o tempo, isto é, as flutuações aumentam ou diminuem com o passar do tempo, indicando que a variância está se alterando. Para detectar a não-estacionariedade de uma série, o comportamento temporal pode ser analisado graficamente, buscando padrões (a) e (b) ou, então, aplicando os testes estatísticos de raiz unitária (Werner e Ribeiro, 2003).

Modelo Auto-Regressivo Integrado de Média Móvel

Comumente, séries podem não apresentar média e variância constantes, sendo assim, trechos aleatórios da série comportam-se de maneira distinta caracterizando-a como uma série não estacionária. Algumas vezes é possível transformar uma série não estacionária em estacionária por meio de sucessivas diferenças entre suas observações. A primeira e segunda diferença de uma série são demonstradas pelas Equações 4.34 e 4.35, respectivamente.

$$\nabla Z_t = Z_t - Z_{t-1} \tag{4.34}$$

$$\nabla^2 Z_t = \nabla[\nabla Z_t] = \nabla[Z_t - Z_{t-1}] = Z_t - 2Z_{t-1} - Z_{t-2} \tag{4.35}$$

A quantidade de diferenças necessárias para que uma série torne-se estacionária é chamada de ordem de integração. Para adicionar a ordem de integração a um modelo assume-se que $w_t = \nabla^d Z_t$ e tem-se a Equação 4.36.

$$w_t = \phi_1 w_{t-1} + \ldots + \phi_p w_{t-p} + \epsilon_t - \theta_1 \epsilon_{t-1} - \ldots - \theta_q \epsilon_{t-q} \tag{4.36}$$

A Equação 4.36 representa o modelo *Auto-Regressivo Integrado Média Móvel (ARIMA)* ou Modelo Auto-Regressivo Integrado de Média Móvel com parâmetros *p, d, q*, ou simplesmente, *ARIMA(p,d,q)*. Os parâmetros p e d são análogos aos parâmetros do modelo *ARMA*, já o parâmetro d de integração corresponde a quantidade de diferenças necessárias para estacionar a série. Este modelo pode ser reescrito utilizando-se o operador de defasagem B, como mostrado pela Equação 4.37.mente, *ARIMA(p,d,q)*. Os parâmetros p e d são análogos aos parâmetros do modelo *ARMA*, já o parâmetro d de integração corresponde a quantidade de diferenças necessárias para estacionar a série. Este modelo pode ser reescrito utilizando-se o operador de defasagem B, como mostrado pela Equação 4.37.

$$\begin{aligned}
(1 - \phi_1 B - \ldots - \phi_p B^p) w_t &= (1 - \theta_1 B - \ldots - \theta_q B^q)\epsilon_t \\
com\ w_t &= (1 - B)^d Z_t \\
\Phi(B)(1 - B)^d Z_t &= \Theta(B)\epsilon_t
\end{aligned} \tag{4.37}$$

O modelo *ARIMA* é ainda mais geral que o modelo *ARMA*. Com o *ARIMA* é possível representar tantos séries estacionárias quanto não estacionárias, sendo que, para muitos autores, o método *Box-Jenkins* resume-se ao *ARIMA*.

4.2.1.3 Modelo Sazonal

O método *Box-Jenkins* ainda possui um modelo que explora a sazonalidade das observações, ou seja, observações que apresentam certas semelhança com outras observações em intervalos constantes de tempo. O modelo de *Box-Jenkins* que trabalha com séries temporais sazonais é o modelo *Seasonal Auto-Regressive Integrated Moving Average (SARIMA)* ou Auto-Regressivo Integrado de Média Móvel Sazonal.

O modelo *SARIMA* é composto de uma parte sazonal e uma parte não sazonal. A parte sazonal possui parâmetros $(P,D,Q)_s$, enquanto a parte não sazonal possui parâmetros (p,d,q). A Equação 4.38 mostra o modelo *SARIMA* mais geral.

$$\begin{aligned} (1-\phi_1 B - \ldots - \phi_p B^p)(1-\Phi_1 B^s - \ldots - \Phi_P B^{Ps}) \\ (1-B)^d(1-B^s)^D Z_t = (1-\theta_1 B - \ldots - \theta_q B^q) \\ (1-\Theta_1 B^s - \ldots - \theta_Q B^{Qs})\epsilon_t \end{aligned} \qquad (4.38)$$

Pela Equação 4.38 tem-se que:

- $(1-\phi_1 B - \ldots - \phi_p B^p)$ é a parte auto-regressiva não sazonal com ordem p;
- $(1-\Phi_1 B^s - \ldots - \Phi_P B^{Ps})$ é a parte auto-regressiva sazonal com ordem P e estação sazonal s.;
- $(1-B)^d$ é parte de integração não sazonal com ordem d;
- $(1-B^s)^D$ é a parte de integração sazonal com ordem D e estação sazonal s.;

- $(1 - \theta_1 B - \ldots - \theta_q B^q)$ é a parte de média móvel não sazonal com ordem q;
- $(1 - \Theta_1 B^s - \ldots - \theta_Q B^{Qs})$ é a parte sazonal de média móvel com ordem Q e estação sazonal s.

4.2.2 Etapas de modelagem Box-Jenkins

Para a escolha de um modelo adequado ao conjunto de dados observados, Box e Jenkins (1976) definem uma abordagem que consiste de 3 passos iterativos:

i. **Identificação:** Com base nos dados históricos da série, esta etapa tem como objetivo identificar quais modelos descrevem o comportamento da série de maneira mais adequada. Tal processo de identificação leva em conta os comportamentos da função de auto-correlação e da *Função de Auto-correlação Parcial (PACF)* ou função de auto-correlação parcial, mostradas pelas Equações 4.39 e 4.40, respectivamente.

$$\hat{\rho}_k = \frac{\frac{1}{N-k}\sum_{t=1}^{N-k}(x_t - \overline{x})(x_{t+k} - \overline{x})}{\frac{1}{N} - \sum_{t=1}^{N}(x_t - \overline{x})^2} \qquad k = 0, 1, \ldots, N/4 \tag{4.39}$$

Segundo Johnson e Montgomery (1974), para que a função de auto-correlação apresente resultados úteis, é necessário que o número de coeficientes de autocorrelação calculados tenha, no mínimo, 25% do tamanho da amostra. Calculados os coeficientes de auto-correlação obtêm-se os coeficientes da auto-correlação parcial resolvendo como mostrado na Equação 4.40 para cada um dos coeficientes resultantes da auto-correlação.

$$\hat{\rho} = \theta_{k1}\hat{\rho}_{j-1} + \theta_{k2}\hat{\rho}_{j-2} + \ldots + \theta_{kk}\hat{\rho}_{j-k} \qquad j = 1, 2, \ldots, k \tag{4.40}$$

Calculados os coefic

ii. **Estimação:** Após a etapa de identificação passa-se para etapa de estimação, na qual os parâmetros ϕ do modelo auto-regressivo, os parâmetros θ do modelo de

Tabela 4.5: Padrões de Auto-Correlação e Auto-Correlação Parcial.

Modelo	**Auto-correlação**	**Auto-correlação parcial**
AR(p)	queda exponencial	θ_{pp} não zero
MA(q)	ρ_q não zero	queda exponencial
ARMA(p,q)	queda exponencial; sinoidal amortecida depois de (q-p)	queda exponencial; sinoidal amortecida depois de (p-q)

Fonte: Johnson e Montgomery (1974)

médias móveis e a variância de ϵ_t são estimados. O método de mínimos quadrados é bastante utilizado para estimativas destes parâmetros, no entanto, outros métodos podem ser empregados nesta etapa.

iii. **Verificação:** Consiste em verificar se o modelo é ou não apropriado a amostra de dados. Uma maneira de realizar a verificação é examinando o comportamento da função de auto-correlação dos resíduos do modelo. Caso esta auto-correlação se apresente como um passeio aleatório, ou seja, diferentes dos padrões da Tabela 4.5, de valores próximos a zero, o modelo pode ser considerado adequado.

No caso do modelo mostrar-se inadequado, as etapas devem ser repetidas até que um modelo adequado seja encontrado. Caso mais de um modelo mostre-se adequado a uma mesma amostra, existem métodos como a máxima verossimilhança e *Critério de Informação de Akaike (AIC)* ou o critério de informação de*Akaike* que permitem escolher entre estes modelos.

Capítulo 5

Arquitetura de Proxy com Prioridades de Serviços para Chamadas Remotas de Procedimento de Tempo Real

Este capítulo apresenta a arquitetura de *proxy* com prioridades de serviços para *RT-RPCs*. O objetivo da arquitetura é privilegiar a transmissão das *RPCs* de maior prioridade e menor *deadline*, de maneira eficiente e racional, sem desconsiderar as demais chamadas com características diferentes. Foi desenvolvida uma arquitetura adaptável que gerencia os recursos disponíveis de maneira inteligente e auto-ajustável.

Para poder gerir os recursos, é necessário que a arquitetura conheça o comportamento do ambiente de rede no qual está inserido, ou seja, conheça os recursos exigidos pelo ambiente. No entanto, para uma alocação de recursos mais eficiente é fundamental que esta ocorra em momentos apropriados e que, se possível, seja antecipada às suas necessidades. Sendo assim, a alocação de recursos pode ser realizada num momento anterior a sua necessidade, em que o *proxy* não está sobrecarregado com o tráfego de mensagens *RPCs*, evitando que o mesmo se ocupe da alocação somente quando os recursos forem necessários, ou seja, quando submetido a fluxos maiores.

Mesmo com a antecipação da alocação dos recursos do *proxy*, a utilização de políticas de comportamento garante que os recursos sejam disponibilizados de maneira suficiente para atender a demanda das *RPCs* de maior prioridade, sem que gerem um impacto acentuado naquelas de menor prioridade.

As políticas permitem controlar o comportamento interno dos módulos da arquitetura e são escolhidas com base nos estados futuros previstos do ambiente de rede. A escolha inteligente de políticas corretas garante um comportamento adequado aos estados atuais e futuros do ambiente de rede, aumentando a eficiência e diminuindo o desperdício de recurso. A Figura 5.1 ilustra a arquitetura proposta neste trabalho.

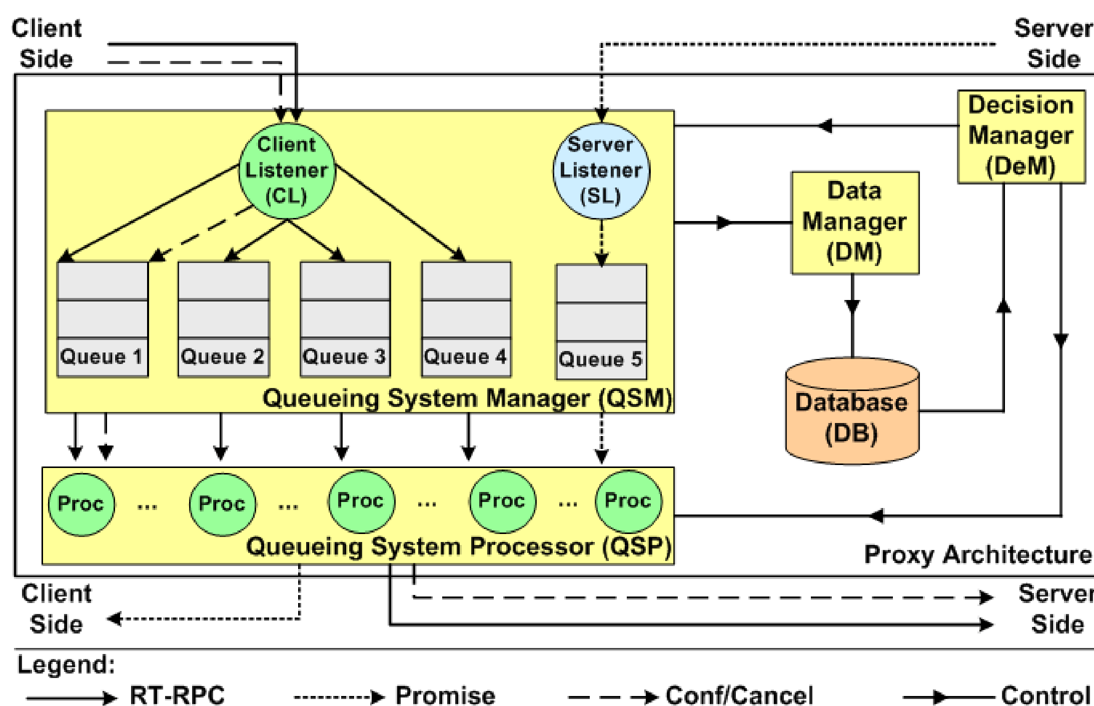


Figura 5.1: Arquitetura do Proxy com QoS para RT-RPCs.

A arquitetura ilustrada na Figura 5.1 define vários módulos que concorrem para o objetivo geral de privilegiar as *RT-RPCs* de maior prioridade. Cada um dos módulos tem funcionalidade bem definida e desempenha um papel fundamental no propósito do *proxy*. As próximas seções detalham esses módulos e discursam sobre uma implementação.

5.1 Queuing System Manager

O *Queueing System Manager (QSM)* é o módulo da arquitetura responsável por receber todo tráfego de mensagens que passa pelo *proxy* e organizá-lo. Este módulo recebe todo tráfego do *lado do cliente* e do *lado do servidor* do *proxy* e organiza-o em estruturas de fila chamadas *Queue*, numeradas de 1 (um) a 5 (cinco). As estruturas *Queue* nada mais são do que filas que armazenam mensagens do protocolo apresentado na seção 3.5. Estas filas são independentes e podem apresentar comportamento distinto umas das outras.

Além de organizar o tráfego, o *QSM* ainda organiza as estatísticas de tráfego capturadas durante sua execução. As estatísticas envolvem:

- *tempo de processamento*: tempo que o *proxy* leva para processar o encaminhamento de uma mensagem retirada de uma das filas para uma das saídas;
- *volume de mensagens*: quantidade de *RT-RPCs* que passaram pelo *proxy* no último intervalo de tempo;
- *confirmação de execução*: quantidade de execuções que ocorreram sem que o *deadline* expirasse e cancelasse a execução;
- *cancelamento de execução*: quantidade de execuções que excederam os requisitos de tempo individuais de cada *RT-RPC*.

Estas estatísticas serão utilizadas posteriormente para tomada de decisão com relação a alocação dos recursos e para adequação do *proxy* ao ambiente de rede no qual está inserido, por meio das políticas de comportamento.

Os módulos principais que compõem o *QSM* e suas funções são detalhados nas subseções seguintes.

5.1.1 Client Listener

O *Ouvidor Cliente (CL)* recebe todo tráfego originado no *lado do cliente* e organiza em 4 (quatro) estruturas de fila. Cada estrutura de fila é responsável por um grupo de prioridades específico.

Neste trabalho utiliza-se o protocolo de aplicação para chamadas remotas de procedimento de tempo real proposto por Villela (2001). As mensagens desse protocolo são divididas em 33 (trinta e três) níveis diferentes de prioridades, que corresponde do intervalo entre os inteiros 0 (zero) e 32 (trinta e dois).

Cada nova mensagem, que encapsula uma *RT-RPC* e chega ao *CL*, tem seu campo de prioridade verificado para identificar o grupo ao qual a mesma pertence. A divisão das mensagens em grupos diferentes leva em consideração apenas a prioridade da mesma, sem considerar do tempo de *deadline*. O *deadline* da mensagem pode ser considerado para inserção da mensagem na fila do grupo ao qual a mesma pertence, no entanto, isso depende da política de inserção a ser aplicada sobre a fila e que será melhor discutido na seção 5.4.3. Convenientemente, as prioridade são divididas em 4 (quatro) grupos de mesmo tamanho:

- *Prioridade baixa*, para mensagens *RT-RPCs* com prioridade de 23 (vinte e três) a 32 (trinta e dois);
- *Prioridade média-baixa*, para mensagens com valores de 15 (quinze) a 22 (vinte e dois);
- *Prioridade média-alta*, para mensagens com valores de 8 (oito) a 14 (quatorze);
- *Prioridade alta*, para mensagens com prioridades de 0 (zero) a 7 (sete).

Com relação as prioridades, a divisão em grupos homogêneos é apenas uma abordagem convenientemente adotada, outras abordagens com características diferentes de divisão poderiam e deverão ser abordadas no futuro.

Além das mensagens *RPCs*, o *CL* ainda responsabiliza-se por organizar as mensagens que encapsulam pedidos de **confirmação** e **cancelamento**. A validade de uma chamada remota de procedimento de tempo real leva em conta tanto a corretitude da execução quanto o cumprimento, ou não, dos requisitos temporais (*i.e. deadline*). Sendo assim, independente da corretitude, ou não, da execução, é fundamental que esta ocorra no tempo que lhe é permitido. As mensagens de *confirmação* e *cancelamento* são enviadas para o executor da chamada confirmando os resultados de execução, quando a chamada foi executada dentro do *deadline*, ou cancelando os resultados da execução, quando o retorno da execução retorna depois do vencimento do *deadline*.

As mensagens de *confirmação* e *cancelamento* não possuem prioridade, tampouco requisitos temporais, e sendo assim, para evitar que o tráfego de mensagens de confirmação e cancelamento interfira no de mensagens *RT-RPC*, as mensagens de *confirmação* e *cancelamento* são inseridas na estrutura de fila do grupo de prioridade baixa, com a política *FCFS*.

Cada uma das quatro filas destinadas a organizarem as mensagens do *lado do cliente* é independente das demais e possui política própria de inserção. A distinção entre as políticas de inserção é necessária, porque as filas podem estar sujeitas a quantidades diferentes de chegada e volume de mensagens. No entanto, suas políticas iniciais de inserção privilegiam mensagens com maior prioridade e menor *deadline*. A questão das políticas de inserção em filas é melhor discutida na seção 5.4.

5.1.2 Server Listener

O *Server Listener (SL)* é responsável pelo tráfego que entra no *proxy* pelo *lado do servidor*. Estas são mensagens do tipo *Promise*, ou seja, armazenam os valores resultantes da execução de uma *RT-RPC*. As mensagens do tipo *Promise* não possuem prioridades ou requisitos temporais, assim como as mensagens de confirmação e cancelamento, o que permite a utilização de uma única estrutura de fila, a *Queue* 5 (cinco), para organizá-las.

Ainda pelo fato das mensagens *Promise* não possuírem requisitos temporais, a política de inserção inicialmente utilizada é a *FCFS*, mas políticas alternativas a esta podem ser utilizadas, como, por exemplo, a inserção priorizando as mensagens *Promise* que retornam valores de *RT-RPCs* de menor *deadline*. No entanto, para implementar uma política desse tipo é necessário que sejam feitas alterações na implementação do *SL*, e que este passe a manter um registro com os identificadores de mensagens *RPCs* com requisitos temporais mais rígidos. Este trabalho ainda não implementa essa característica.

5.2 Queueing System Processor

O *Queueing System Processor (QSP)* é responsável pela extração das mensagens organizadas e inseridas nas estruturas *Queue* do *QSM* e pelo direcionamento para linha de saída correta, seja ela o *lado do servidor* ou o *lado do cliente*.

O *QSP* aloca os processadores lógicos de filas para as estruturas *Queue* do *QSM*, sendo que esses processadores são linhas de execução, independentes e reutilizáveis, alocadas com a finalidade de atender a demanda de tráfego de mensagens que passa pelo *proxy*. Uma estrutura de fila de um grupo de prioridades pode, ao mesmo tempo, ter mais de um processador responsável por sua extração. Uma vez alocado, um processador só deve ser desalocado se o estado atual e os estados futuros, previstos para o ambiente de rede, indicarem quantidades bastantes reduzidas de tráfego em relação ao número de processadores disponíveis, ou a necessidade de alguém com maior prioridade.

Quando em condições de congestionamento em alguma das filas de prioridade, o *QSP* pode optar por realocar processadores de uma fila de menor congestionamento para outra que necessite de mais processadores.

Os processadores responsáveis por uma fila de prioridades devem ser em número suficiente para garantir que nenhuma chamada permaneça na fila até a vencimento do *deadline*. Dessa forma, uma das tarefas mais importantes da arquitetura é determinar qual o número ideal de processadores para cada fila.

Pelas características e disponibilidade das informações, o *proxy* determina o número adequado de processadores com base no seguinte:

- Quantidade de mensagens que um processador consegue extrair e direcionar mensagens de uma fila por unidade de tempo;
- Tamanho atual da fila em questão;
- Tamanho previsto da fila após um intervalo de tempo, ou seja, quanto espera-se que a fila aumente em um período definido de tempo;
- *Deadline* das mensagens que esperam por transmissão na fila, principalmente, o menor.

Cada processador é capaz de processar um número limitado de mensagens por unidade de tempo, e sendo assim, quanto maior o número de mensagens, maior deve ser a quantidade de processadores alocados suficientes para atender a demanda atual. Para este trabalho limitou-se o número de processadores em 1024 baseando-se em experimentos que são detalhados no capítulo 6.

As previsões permitem a alocação prévia de recursos que serão necessários para o atendimento de uma demanda iminente, possibilitando a adaptação e a preparação dos recursos do *proxy* para um futuro breve. As vantagens da prévia alocação de recursos num momento de baixa demanda evita que o *proxy* tenha que disprender esforços com a alocação durante o período de maior demanda, o que pode ser comprometedor quanto a requisitos temporais.

Os processadores responsáveis por uma mesma fila seguem políticas idênticas de extração. No entanto, o mesmo não é garantido para aqueles que atuam em filas diferentes e estes podem seguir políticas independentes (*e.g.* É possível que um os processadores de um grupo atuem em linhas de execução com maior prioridade que os de outro grupo de prioridade, ou então que processadores de um grupo sejam impedidos de realizarem

extrações enquanto a fila de outro grupo de prioridade maior não estiver completamente vazia). A seção 5.4.3 discute melhor as políticas de extração.

5.3 Data Manager

O *Gerenciador de Dados (DM)* recebe as informações sobre estatísticas arrecadadas pelo *QSM*, sumariza e armazena nas estruturas adequadas para persisti-las no *Base de Dados (DB)*. Essas informações devem estar disponíveis para o acesso do *Decision Manager* que as utiliza para o processo de tomada de decisão. Por meio da ação do *DM* que o *Decision Manager* obtém informações sobre os estados passados do ambiente de rede, uma vez que estas representam o volume do tráfego e as características das mensagens que passaram pelo *proxy*.

O *DM* atua em intervalos bem definidos de tempo. É necessário que o *QSM* arrecade uma quantidade suficiente de informação que justifique sua persistência.

5.4 Decision Manager

A maior parte da complexidade da arquitetura localiza-se *Gerenciador de Decisão (DeM)*. Enquanto os outros módulos desempenham funções automáticas, o *DeM* é responsável por toda tomada de decisão que acontece dentro da arquitetura.

O *DeM* determina a reação dos demais módulos aos estados do ambiente de rede, por meio da escolha de políticas de comportamento. O processo de tomada de decisão envolve o entendimento dos estados anteriores e do estado atual experimentados, e dos recursos disponíveis para o *proxy* atender a demanda de tráfego. Com tais informações disponíveis o *proxy* pode determinar quais são as políticas adequadas para cada um dos módulos da arquitetura, bem como os recursos que precisam ser alocados.

As decisões tomadas pelo *DeM* são repassadas aos módulos correspondentes para determinar o comportamento dos mesmos em relação aos estados futuros. O *Decision Manager*

é composto por 2 módulos principais: O *Prediction System (PS)* e o *Decision System (DS)*. Cada um com uma funcionalidade específica no processo de decisão. A Figura 5.2 ilustra o módulo *DeM* e seus componentes.

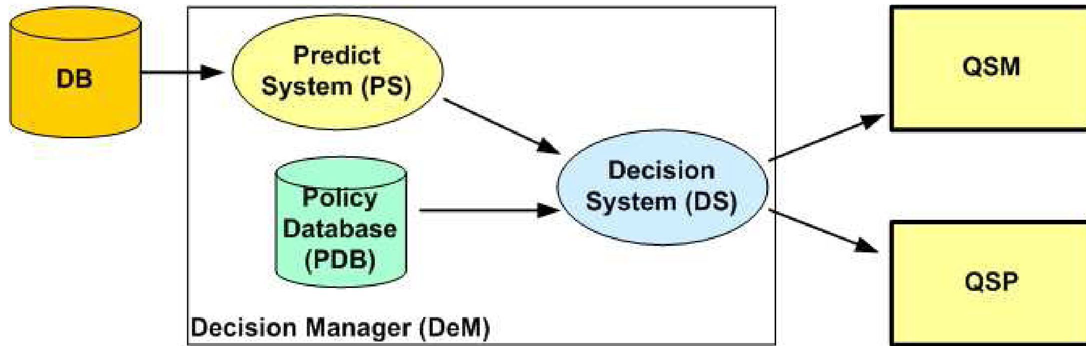


Figura 5.2: Módulos do Decision Manager.

Os módulos do *DeM* e suas funcionalidades são detalhados nas próximas subseções.

5.4.1 Prediction System

O *Sistema de Previsão (PS)* realiza previsões com o intuito de antecipar os estados futuros do ambiente de rede. Com base nos estados passados (*i.e.* volume e características das mensagens *RT-RPCs* que passaram pelo *proxy*), o *PS* utiliza modelos de séries temporais adequados para realizar previsões sobre a características e a quantidade de mensagens *RT-RPCs* para qual o *proxy* deve estar preparado. As previsões fornecem base às tomadas de decisões e permitem que o *proxy* adeque-se a um futuro breve do seu ambiente, alocando e desalocando recursos necessários, e determinando de que maneira cada módulo deve agir em relação ao novo estado.

Em intervalos de tempo bem definido, o *PS* recupera as informações, outrora, armazenadas no *DB*, para realizar suas previsões. O *DB* persiste informações sobre o volume de mensagens *RT-RPC* para cada grupo de prioridade, e de confirmação/cancelamento que passaram pelo *proxy*. As informações sobre o volume de mensagens experimentado pelo *proxy* são utilizadas como estatísticas dos estados passados do ambiente.

Os estados passados dizem muito sobre o que ainda pode acontecer, sendo que é possível que estes estados tenham relação uns com os outros durante as sucessivas observações. Uma vez identificado algum relacionamento entre eles, pode-se dispor de um método de séries temporais para modelar este comportamento e, assim, realizar previsões.

O método escolhido neste trabalho foi o método *Box-Jenkins* (Box e Jenkins, 1976). Este método reúne os modelos de *Box-Jenkins* que, ao contrário dos métodos baseados em suavização exponencial, exploram a possível dependência entre os dados observados. O método *Box-Jenkins* estabelece modelos para conjuntos de dados que apresentam comportamento estacionário, não estacionário ou sazonal. Estes modelos podem ser combinados para abranger o maior número possível de séries temporais, sem a necessidade de métodos adicionais para estacionamento das séries (Morettin e Toloi, 1981a).

5.4.1.1 Ajuste do Prediction System

Por sua funcionalidade preditiva, o *PS* é o módulo da arquitetura que deve ter conhecimento prévio do ambiente de rede do *proxy*, uma vez que previsões confiáveis só podem ser obtidas se o comportamento do ambiente for conhecido e modelado corretamente.

Para que o *PS* tenha condições de realizar previsões, os modelos de previsão devem estar implementados e disponíveis, o que só se torna possível com uma análise preliminar do tráfego de mensagens. Basicamente, o ajuste do *PS* consiste da exploração do ambiente de rede para arrecadação de informações sobre seu comportamento, e posterior modelagem deste comportamento.

Para a coleta de informações sobre o tráfego de mensagens, utiliza-se uma versão especial do *proxy*, cujo intuito é justamente arrecadar as informações sobre comportamento, e que aqui é denominado *proxy* coletor. Esta versão especial implementa apenas os módulos necessários para a coleta e o armazenamento das informações do ambiente de rede. A Figura 5.3 ilustra os módulos que formam o *proxy* coletor.

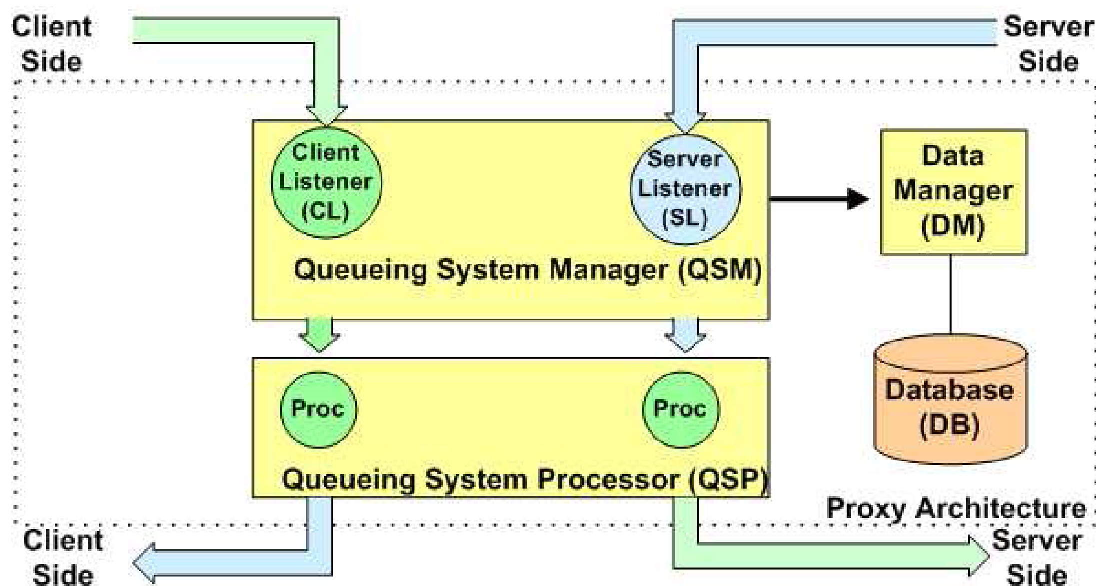


Figura 5.3: Arquitetura do Proxy Coletor.

O *proxy* coletor atua de maneira transparente, ou seja, sua ação e existência são desconhecidos pelas aplicações que se comunicam através da rede. Este *proxy* redireciona todo o fluxo vindo do *lado do cliente* para o *lado do servidor* e vice-versa, sem qualquer alteração na ordem das mensagens. Não são utilizadas filas porque não há reorganização das mensagens, e apenas 2 (dois) processadores são alocados para o *QSM*, um para o *CL* e outro para o *SL*.

As observações do volume de mensagens de cada grupo de prioridade são repassadas ao *DM*, que então sumariza, organiza e armazena as informações no *DB*. São essas informações que permitem que seja modelado o comportamento da rede. A partir de uma análise desses dados, é possível identificar qual o melhor modelo e quais os melhores parâmetros para que sejam realizadas previsões mais próximas possíveis dos valores reais observados.

O *proxy* coletor é executado na rede com o intuito de colher uma quantidade suficiente de informações que descrevam aquele ambiente. Os passos para o ajuste do *PS* são:

Passo 1: Coleta de informações sobre o ambiente de rede.

Execução do *proxy* coletor sob um fluxo real de mensagens para as quais se pretende realizar previsões. Durante este período, o *proxy* arrecada informações sobre a quantidade e características das mensagens que trafegam pela rede, e avalia as reações do *proxy* quando submetido a diferentes tráfegos de mensagens.

O tempo pelo qual os dados são coletados deve ser suficiente para um entendimento completo do ambiente, e pode variar de uma rede para outra. Quanto maior a quantidade de informações da rede, mais adequado será o modelo de previsão. Ao final da coleta, o comportamento real da rede é representado dentro do *DB* e está disponível para análise.

Passo 2: Modelagem dos sistemas de previsão.

Utilização das informações da base de dados *DB* para identificação e construção do modelo de previsão.

Depois de um comportamento real da rede estar disponível no *DB* é o momento de buscar relações que permitam modelar tal comportamento. A modelagem segue os passos propostos na seção 4.2.2. Para cada grupo de prioridade deve ser criado um modelo de previsão, que será responsável por antecipar o comportamento das mensagens pertencentes a um grupo em específico.

A fase de modelagem requer o auxílio externo de algum pacote estatístico para cálculo dos valores das funções de auto-correlação e auto-correlação parcial das séries temporais (Morettin e Toloi, 1981b). Calculadas as funções de auto-correlação e auto-correlação parcial devem ser gerados os gráficos dessas funções para que, baseado na tabela exposta em 4.2.1, seja possível identificar o modelo que melhor representa a série.

Neste trabalho utilizou-se o pacote estatístico *R* (RProject, 2007). Esta ferramenta é bastante completa e permite que, a partir de um série temporal de entrada, sejam calculados os valores para as funções e sejam gerados os gráficos correspondentes. A partir dos gráficos é possível identificar qual o tipo e o comportamento da série de acordo com as definições de Box e Jenkins (1976).

Passo 3: Inserção do modelo no *PS*.

Uma vez que o modelo foi identificado e verificado, o próximo passo é inserir o modelo já com os devidos parâmetros no *PS*. Este passo envolve a entrada dos parâmetros estimados no passo anterior para dentro da arquitetura. Dessa forma, o *proxy* pode ser posto em execução e já pode realizar previsões.

Uma vez que as previsões podem, ocasionalmente, falhar, o modelo é sempre realimentado com os valores reais observados para corrigir as falhas das previsões. Isso garante que o modelo sempre buscará adaptar-se às variações que ocorrerem entre previsões e valores reais.

5.4.2 Decision System

O *Sistema de Decisão (DS)* é o administrador da tomada de decisões do *proxy*. Dos módulos do *Decision Manager*, o *DS* executa o papel fundamental ao qual se propõe o *DeM*, uma vez que recebe os dados de entrada, executa a tomada de decisão e repassa a mesma aos demais módulos do *proxy*.

Este módulo implementa um sistema de decisão autônomo, cujas decisões são baseadas na reunião de informações geradas por outros componentes da arquitetura. As decisões envolvem diretamente a alocação racional dos recursos do *proxy* e escolha de políticas de comportamento para os módulos *QSM* e *QSP*.

Para o processo de decisão, o *DS* necessita de duas importantes entradas: As previsões geradas pelo *PS* e a base de conhecimento de políticas, *Policy Database*. O *PS* realiza as previsões e passa as mesmas para o *DS* que, em posse dessas informações, consegue projetar como possivelmente as aplicações de tempo real irão se comportar. Baseado nesses comportamentos, o *DS* estima o número de mensagens e as características das mesmas.

Após receber as previsões, o *DS*, através de uma máquina de inferência, consulta a *Policy Database* para saber quais são as políticas de comportamento são mais adequadas para os futuros estados, repassando-as para os módulos *QSM* e *QSP*.

Ao receberem as políticas de comportamento o *QSM* e *QSP* imediatamente tomam as atitudes necessárias para adaptarem-se às políticas e, conseqüentemente, ao ambiente de rede.

A opção de utilizar uma máquina de inferência lógica para consulta a *Policy Database* é justificada pelo fato de máquinas de inferência serem adequadas, e amplamente utilizadas, na implementação de sistemas inteligentes como os sistemas de decisão e os *Sistemas de Suporte à Decisão (SSD)*.

As máquinas de inferência implementam algoritmos otimizados e mantêm separados os mecanismos busca da base de conhecimento. A separação entre base de conhecimento e mecanismos de busca permite que a base de conhecimento seja dinâmica e independente da implementação dos mecanismos, permitindo que a base seja atualizada constantemente.

Neste trabalho o mecanismo de inferência utilizado é o implementado pela ferramenta GNU Prolog (Diaz, 2007), uma implementação de compilador para linguagem *Prolog* do paradigma de programação lógico.

5.4.3 Políticas e Policy Database

As políticas de comportamento são todas armazenadas na *Base de Dados de Políticas (PDB)*. O que neste trabalho vem sendo tratado como políticas de comportamento são entendidas como imposições de execução que devem ser seguidas por alguns módulos da arquitetura. Tais módulos têm funcionalidades específicas (*i.e.* *QSM* e *QSP*) e, sendo assim, cumprem exigências diferentes de comportamento.

A *PDB* armazena regras que são consultadas por uma máquina de inferência lógica. Para consultar essas regras são passados parâmetros de entrada, que permitem selecionar, dentre as diversas regras, a melhor política a ser aplicada. A Figura 5.4 ilustra um trecho da *PDB* implementado em *Prolog*.

Para o *QSM*, as políticas de comportamento estabelecem quais são as políticas de inserção a serem utilizadas por cada uma das estruturas *Queue*. As filas destinadas ao *CL*

```
aloca_proc( QPA, QPN, QPT, TFP, TFA, QM, MD):-
                            soma( TFP, TFA, NUM),
                            mult( QM, MD, DEN),
                            div ( NUM, DEN, QPN),
                            proc_alocados(QTP, QPN, QPA).

proc_alocados( QPT, QPN, QPA) :- QPN =< QPT, !, QPA is QPN.
proc_alocados( QTP, QPN, QPA) :- QPA is QPT.
```

Figura 5.4: Trecho da Policy Database.

limitam-se a grupos de prioridades diferentes e, conseqüentemente, a tráfegos diferentes de mensagens. Estas filas podem apresentar comportamentos distintos umas das outras. Os possíveis comportamentos de inserção para estas filas são baseadas em :

i. **Momento de chegada**

   A inserção com base na chegada não considera as características (*i.e.* prioridade e *deadline*) das mensagens, organizando-as com base apenas no instante de chegada. Para isso usa-se a inserção *FCFS* ou *First In, First Out (FIFO)*, em que as mensagens que acabam de chegar são colocadas no final da fila e são as últimas a serem tratadas.

   Na maioria dos modelos de fila tradicionais, este é o modo padrão de inserção adotado, uma vez que se trata de uma inserção rápida e direta. No entanto, em se tratando de mensagens *RT-RPC* com prioridades e requisitos de tempo, esta política pode ser usada apenas sob as seguintes restrições:

   - o volume de mensagens é reduzido que passa pelo *proxy*,
   - as filas estão pouco ocupadas,
   - os recursos alocados são suficientes para atender a demanda atual sem perdas,
   - as execuções de chamadas são bem sucedidas,
   - os *deadlines* são grandes mesmo para as mensagens de prioridade alta,
   - não há previsão de aumento de tráfego.

Caso todos os itens acima sejam verificados, o *FCFS* pode ser aplicado como política de inserção para determinada fila. Qualquer desvio nessas regras implica na escolha de outra prática de inserção.

ii. **Prioridade**

A política baseada em prioridade não considera o momento em que a mensagem entrou no sistema, mas sim qual a prioridade da mensagem em relação as que já estão na fila. A mensagem é organizada na fila de acordo com sua prioridade, sendo inserida na frente de mensagens com menor prioridade e atrás de mensagens com prioridade maior.

Esta política é rígida quanto a colocação da mensagem em relação à prioridade, sendo ideal quando:

- o volume de mensagens é grande,
- os recursos alocados são suficientes para atender a demanda atual,
- as filas estão bastante ocupadas,
- os *deadlines* são grandes,
- as execuções de chamadas são bem sucedidas,
- a previsão é de que o tráfego permanecerá constante ou aumentará.

No entanto, a inserção por prioridade não leva em conta os requisitos de tempo da mensagem, e permite que mensagens com menores requisitos de tempo sejam privilegiadas em relação a mensagens mais rígidas, o que pode incorrer na expiração de mensagens por tempo em fila.

iii. ***Deadline***

Assim como a política de inserção com base em prioridade, a política com base no *deadline* não considera o momento em que a mensagem entrou no sistema. São

analisados os *deadlines* das mensagens dentro da fila e a mensagem recém-chegada é inserida na frente de mensagens com menor *deadline* e atrás de mensagens com *deadline* maior. Esta política pode ser usada apenas sob as seguintes restrições:

- o volume de mensagens é grande,
- os recursos alocados não são suficientes para atender a demanda atual,
- as filas estão bastante ocupadas,
- os *deadlines* são pequenos,
- a previsão é de que o tráfego permanecerá constante ou aumentará,
- falhas ocorrem devido a expiramento de *deadline*.

iv. **Prioridade e *deadline***

Esta política reúne características das inserções com base em prioridade e *deadline*. Primeiramente, as mensagens são organizadas por prioridade. Caso existam mensagens com prioridades iguais inseridas na fila, a mensagem é inserida, junto as mensagens de mesma prioridade, com base no *deadline*.

A inserção com base nessa política é computacionalmente mais complexa que as anteriores, uma vez que a mensagem deve ocupar o lugar correto (*i.e* com base nos requisitos de tempo) dentro de seu grupo de prioridade. Apesar de mais complexa, esta é a política que melhor organiza as mensagens por basear-se em ambas as características da mesma. Esta política é indicada quando:

- o volume de mensagens é grande,
- as prioridades e *deadlines* são bem diferentes,
- os recursos alocados não são suficientes para atender a demanda atual,
- as filas estão bastante ocupadas,
- a previsão é de que o tráfego permanecerá constante ou aumentará,

Apesar de muitas das políticas de inserção apresentarem requisitos em comum, o processo de escolha direta de uma delas não é uma tarefa simples. Assim foi criada a *PDB*, cujo intuito é armazenar as políticas e permitir a fácil alteração com base nas peculiaridades do ambiente de rede.

O *QSP* aloca os processadores lógicos de filas para as estruturas *Queue* do *QSM* com a finalidade de atender a demanda de tráfego de mensagens que passa pelo *proxy*. Os processadores devem ser em número suficiente para garantir que nenhuma chamada permaneça na fila até a expiração do *deadline*. O número de processadores é determinado com base em:

- Quantidade de mensagens que um processador consegue extrair de uma fila por unidade de tempo;
- Tamanho atual da fila em questão;
- Tamanho previsto da fila após um intervalo de tempo;
- *Deadline* das mensagens que esperam por transmissão na fila.

A *PDB* também conta com regras para determinar o número de processadores que devem ser alocados pelo *QSP* para cada uma das filas. Essas regras consideram os requisitos expostos anteriormente e, com base neles, o número ideal de processadores pode ser calculado, como mostrado pela Equação 5.1.

$$Q_p = \frac{T_{fp} + T_{fa}}{Q_{rpc} D} \tag{5.1}$$

Pela Equação 5.1, tem-se:

- Q_p como a quantidade de processadores necessários;
- T_{fp} como tamanho da fila previsto (*i.e.* o quanto espera-se que a fila cresça);

- T_{fa} como tamanho da fila atualmente;

- Q_{rpc} como quantidade de *RT-RPC* processadas por unidade de tempo;

- D como menor *deadline* de uma *RT-RPC* na fila.

Para cada fila é determinado um número de processadores de acordo com a necessidade relativa ao tráfego de mensagens atual e previsto. Quando recebe as ordens do *DeM*, o *QSP* encarrega-se de fazer as devidas alocações para cada uma das filas quando necessário.

A alocação só é necessária se a quantidade de processadores atualmente alocados for menor que a estipulada pelo *DeM*. Nessa situação, somente a diferença entre a quantidade estipulada e os processadores já alocados será alocada de fato.

No caso de redução do tráfego, o *QSP* pode continuar com a mesma quantidade de processadores ou receber uma ordem de desalocação. Uma ordem de desalocação impõe ao *QSP* que reduza a quantidade de processadores para uma determinada fila, sendo útil quando há escassez (*i.e.* os processadores disponíveis não são suficientes para atender uma quantidade ideal calculada) para alguma fila e alguma outra fila possui mais processadores que o suficiente. Outro caso é quando são necessários mais processadores para uma fila de alta prioridade e não há mais disponíveis. Assim deve-se realizar uma desalocação de processadores ativos, penalizando sempre as fila de menor prioridade.

A quantidade total de processadores é limitada, sendo que estes são primeiramente alocados para as filas de maior prioridade. O número máximo de processadores não limita a quantidade máxima que pode ser alocada para cada fila. Caso a quantidade ideal não possa ser alocada, serão alocados a quantidade máxima possível para filas de menor prioridade. Quando a necessidade parte de uma fila de maior prioridade são geradas ordens de desalocação.

A *PDB* é parametrizada, permitindo que seus valores limites possam ser alterados dinamicamente. Neste trabalho, não está previsto o balanceamento na desalocação para que uma fila não fique desprovida de processadores.

Os processadores podem estar alocados para filas de diferentes prioridade, expostas a diferentes tipos de mensagens e que, por isso, devem comportar-se de maneira distinta. Na *PDB* são definidas políticas de comportamento para os processadores já alocados. Essas políticas definem se os processadores de fila menos prioritárias podem ou não extrair mensagens de suas filas enquanto filas de maior prioridade não estiverem vazias. No primeiro caso, as filas de menor prioridade permanecem inertes enquanto uma fila de maior prioridade não estiver vazia, e só começam a ser processadas quando a fila de maior prioridade esvaziar. A política padrão adotada permite a disputa justa entre os processadores das diversas filas.

Políticas alternativas podem ser implementadas como por exemplo a mudança de comportamento baseada na quantidade de mensagens em fila e/ou *deadline*, no entanto, este trabalho ainda não implementa esses tipos de políticas.

Capítulo 6

Estudo de Caso

Neste capítulo é apresentado um estudo de caso para validar a arquitetura proposta. Neste estudo de caso, uma implementação da arquitetura é exposta a um volume variável de tráfego de todos os tipos de mensagens, com diferentes prioridades e *deadlines*, submetendo o *proxy* a maior quantidade e diversidade de estados possíveis.

Para realização dos testes foi desenvolvida uma aplicação responsável por gerar mensagens *RT-RPCs* com diferentes características. Esta aplicação gera volumes variáveis de *RT-RPCs*, resultando em diferentes estados de tráfego na rede e tem o intuito de inundar o *proxy* com diferentes volumes de mensagens, criando um ambiente onde possam ser analisados os comportamentos dos diversos módulos da arquitetura, principalmente, os módulos envolvidos com a tomada de decisão.

O objetivo deste estudo de caso é analisar se, quando submetido a um grande volume de mensagens, o *proxy* realiza boas previsões sobre situações futuras e opta pelas políticas de comportamento mais adequadas.

Como objetivo secundário, o estudo de caso busca comparar os resultados obtidos pela utilização de diferentes políticas. A análise da aplicação das diferentes políticas de comportamento é feita por meio do controle das variáveis envolvidas.

Por variáveis envolvidas, entende-se as variáveis de controle da aplicação teste com relação a quantidade e características das *RT-RPCs*, e as variáveis que controlam as ações do *proxy*. Nas próximas subseções serão detalhados as etapas envolvidas no estudo de caso.

6.1 Desenvolvimento da Aplicação de Teste

A aplicação de testes desenvolvida gera um fluxo variável de *RT-RPCs* para criar, no ambiente de rede, estados em que sejam possíveis analisar as diferentes partes que compõem a arquitetura, principalmente aquelas que envolvem o processo de tomada de decisão.

Esta aplicação é composta de dois módulos com funcionalidades distintas: um módulo cliente e um módulo servidor. O módulo cliente encarrega-se da geração de chamadas e controle dos requisitos de prioridade e tempo das *RT-RPCs*, enquanto que o módulo servidor é responsável pela execução das chamadas e pela organização das estatísticas de execuções bem e mal sucedidas.

A aplicação cliente possui parâmetros de que permitem controlar o número de requisições, bem como os intervalos de prioridade e *deadline*. Além desses parâmetros, a aplicação ainda permite que o usuário defina intervalos entre envio de requisições sucessivas. A Figura 6.1 ilustra a comunicação entre os módulos cliente e servidor.

Como o foco principal dos testes é analisar a validade da arquitetura e não a execução das chamadas, as *RT-RPCs* geradas pela aplicação cliente são bastante simples, sendo que as mesmas são desprovidas de valores de retorno e de argumentos. Já no servidor, a implementação dos procedimentos possui corpo de execução vazio.

A ferramenta desenvolvida em (Villela, 2001) permite a criação de aplicações gráficas com suporte a *RT-RPCs*, porém a ferramenta não dá suporte a geração de código para múltiplas linguagens. Além disso, as diversas camadas de abstração adicionadas pelas classes geradas pela ferramenta tornam a geração de requisições bastante lenta. Nessa

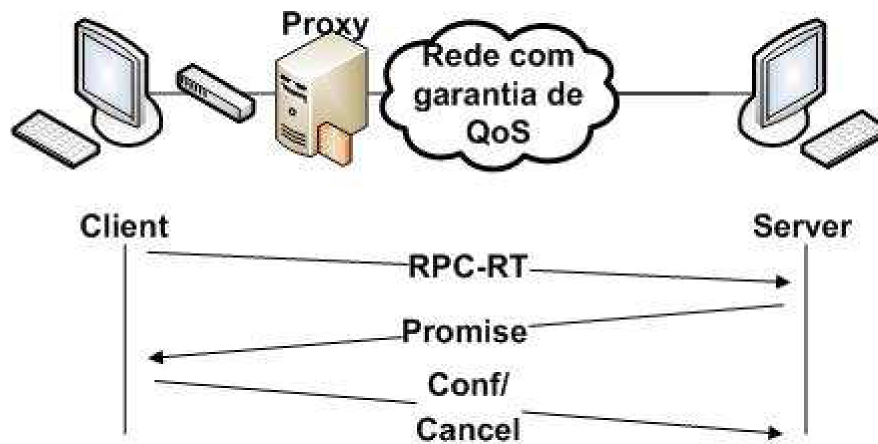


Figura 6.1: Comunicação entre aplicação cliente e servidor.

ferramenta, a aplicação cliente e seus módulos de comunicação são gerados sempre em linguagem *Java*. Outra característica é que a ferramenta também é que apenas os módulos de comunicação da aplicação servidora são gerados, e não a aplicação completa, sendo que o código-fonte desses módulos são gerados sempre na linguagem *C++*.

Para realização dos testes foi gerada uma aplicação na ferramenta supracitada, mas por questões de desempenho e compatibilidade, a aplicação de testes (*i.e.* cliente e servidor), bem como os módulos de comunicação, foram reimplementados em linguagem *C*. As próximas subseções detalham o funcionamento dos módulos cliente e servidor para a aplicação de testes.

6.1.1 Módulo Cliente da Aplicação de Teste

O módulo cliente encarrega-se da geração e controle das *RT-RPCs* com diferentes prioridades e *deadlines*. Este módulo pode iniciar várias linhas de execução independentes de modo a aumentar o número de mensagens a serem enviadas para o servidor.

Para cada mensagem *RT-RPC* enviada, é criado e inserido um registro numa fila de controle interno do cliente. Nessa fila são inseridos os registros de chamadas com menores *deadlines* no início, enquanto os maiores *deadlines* são inseridos próximos do final.

O cliente possui um temporizador que, em intervalos bem definidos, percorre a fila verificando quais *deadlines* expiraram, ou seja, quantas execuções falharam por desrespeito aos requisitos temporais, retirando os registros correspondentes da fila. Existem duas maneiras de um registro ser extraído da fila: por expiração de *deadline* ou quando o resultado correspondente a chamada é retornado após a execução.

Quando a resposta da execução de uma chamada chega ao cliente encapsulada por uma mensagem do tipo *Promise*, este procura pelo respectivo registro na fila. Caso o registro seja encontrado, ele é retirado, e uma mensagem do tipo **confirmação** é enviada ao servidor. No caso contrário uma mensagem do tipo **cancelamento** é enviada. Somente após o recebimento de uma mensagem de confirmação, ou cancelamento, o servidor pode tomar alguma atitude em relação ao que foi alterado durante a execução da chamada.

6.1.2 Módulo Servidor da Aplicação de Teste

O módulo servidor é responsável por receber as mensagens *RT-RPCs*, executar e organizar as estatísticas em casos de confirmação e cancelamento da execução.

O servidor recebe as chamadas vindas do cliente, organiza em uma fila por ordem de chegada e a medida que as execuções são realizadas, mensagens do tipo *Promise*, contendo o resultado da execução, são enviadas ao cliente. Outras políticas de inserção não são tratadas aqui por fugirem do escopo deste trabalho.

Após o envio da *Promise*, o módulo servidor espera que o cliente envie uma mensagem confirmando ou cancelando a execução. Uma mensagem de confirmação é enviada sempre que a soma dos tempos de execução e transmissão não ultrapassem o *deadline*, caso contrário uma mensagem de cancelamento é enviada.

Caso uma mensagem de confirmação seja recebida, o servidor valida a execução do procedimento. Caso uma mensagem de cancelamento seja recebida o servidor volta ao seu estado anterior à execução.

Os procedimentos executados pelo servidor possuem corpo de execução vazio, uma vez que os testes não estão focados na execução dos procedimentos, mas na análise dos módulos da arquitetura. Sendo assim, a confirmação e/ou cancelamento da execução de um procedimento são úteis apenas para fornecer informações sobre a estatísticas de sucesso e falha da execução.

6.2 Análise Preliminar do Comportamento do Ambiente de Rede

A análise preliminar do comportamento do ambiente de rede constitui o primeiro passo para entendimento do ambiente ao qual o *proxy* será submetido. Tal análise permite arrecadar informações sobre a quantidade e características das mensagens que trafegam pela rede, e ainda permite avaliar as reações do *proxy* quando submetido a diferentes tráfego de mensagens. A partir do estudo do comportamento são levantadas as informações necessárias para construção dos modelos de previsão.

Para analisar o comportamento de rede é utilizada uma versão parcial do *proxy* para coleta de dados, descrito em 5.4.1.1. Este *proxy* atua de modo transparente, ou seja, imperceptível às aplicações, e sem que qualquer alteração seja necessárias. Dessa forma, as informações da rede podem ser recolhidas sem qualquer interferência na comunicação. As próximas subseções detalham a coleta e relevância das informações obtidas.

6.3 Tempo de Processamento

Para atender uma demanda de tráfego é preciso que um número suficiente de processadores de fila sejam alocados para cada uma das filas. Quanto mais próximo do ideal for o número de processadores alocados, menor será o aumento da fila de espera.

Uma das medidas necessárias para o cálculo do número ideal de processadores é o tempo que um processador utiliza para processar um mensagem. O processamento de uma mensagem envolve sua extração da fila, o encaminhamento para saída correta e a transmissão da mensagem.

O estudo das observações do comportamento do ambiente de rede permite que sejam feitas medidas dos tempos de processamento utilizados de acordo com a quantidade de mensagens. Das observações é possível extrair várias medidas de síntese como a **média**, a **mediana**, a **moda**, e os valores máximo e mínimo, por exemplo, dos quais se pode distingüir qual destas medida representa melhor o tempo de processamento.

A média constitui o ponto de equilíbrio da amostra e é bastante influenciada por valores anômalos, ou seja, muito diferentes do que estão sendo observados, mesmo que sejam esporádicos. Estes valores podem, equivocadamente, tendenciar a média para cima ou para baixo, neutralizando a influência de outras, talvez muitas, observações que permanecem dentro de um intervalo de normalidade. A média só constitui uma medida expressiva quando acompanhada de uma baixa variância da amostra.

Determinar o número de processadores com base na média é uma estratégia arriscada, uma vez que deixaria de considerar os tempos de processamento maiores que esta, podendo resultar em um número de processadores menor que o ideal.

A mediana divide a amostra em duas partes iguais e não é influenciada por valores anômalos. No entanto, a mediana não considera a amplitude das diferenças individuais entre os valores observados, uma vez que as observações são divididas em conjunto das observações maiores e menores que o valor de mediana. Logo, a mediana das observações como tempo de processamento não é uma escolha viável, uma vez que, obrigatoriamente, metade das observações (*i.e* as observações maiores que a mediana) são desconsiderados. Assim como a mediana, valores de quartis não são interessantes por desconsiderarem parcelas grandes da amostra.

A moda representa o valor mais recorrente no conjunto de observações. Para uma amostra de dados com muitos valores heterogêneos, a moda não é uma boa representação dos dados, já que pode assumir o valor de um pequeno conjunto de observações idênticas, mesmo que estas não representem suficientemente a amostra analisada.

A escolha da moda como tempo de processamento é uma boa escolha se a amostra constitui-se de maneira homogênea, mas para amostras pouco homogêneas a moda pode ser ainda mais arriscada que a média.

Outra possibilidade de representação é optar maior tempo de processamento. O maior valor observado não representa muito sobre a amostra, todavia não penaliza nenhuma das outras observações por estar acima de todos os valores observados. Se o número de processadores for baseado no maior tempo de processamento, então pelo pior caso, o número de processadores alocados e, no mínimo, pouco superior ao ideal.

O maior tempo, necessariamente, não é a maior observação encontrada. Observações muito altas e fora da normalidade podem ser sacadas da amostra, uma vez que podem decorrer de tarefas do sistema operacional ou interrupções do *hardware* em que o *proxy* está executando (*e.g.* acesso a disco pelo sistema operacional). correr de tarefas do sistema operacional ou interrupções do *hardware* em que o *proxy* está executando (*e.g.* acesso a disco pelo sistema operacional).

6.3.1 Medidas de Tempo de Processamento

O tempo de processamento é o somatório dos tempos de extração, redirecionamento de saída e transmissão de uma mensagem. Para medir os tempos de processamento, a aplicação cliente foi parametrizada para inundar o *proxy* com requisições *RT-RPCs*, de forma que as filas do *QSM* nunca estejam vazias.

Cada requisição gera 3 (três) medições de processamento: uma para requisição *RT-RPC*, uma para mensagem *Promise* e uma para mensagem de confirmação ou cancelamento. Essas mensagens possuem tamanhos e características diferentes, e conseqüente-

mente geram tempos distintos de processamento. A Figura 6.2 mostra a variação nos tempos de processamento para 3 (três) dos experimentos realizados. Nesses experimentos não foram considerados os números de chamadas confirmadas ou canceladas, apenas os tempos de processamento foram analisados. Cada teste envolveu o envio de 20000 (vinte mil) requisições.

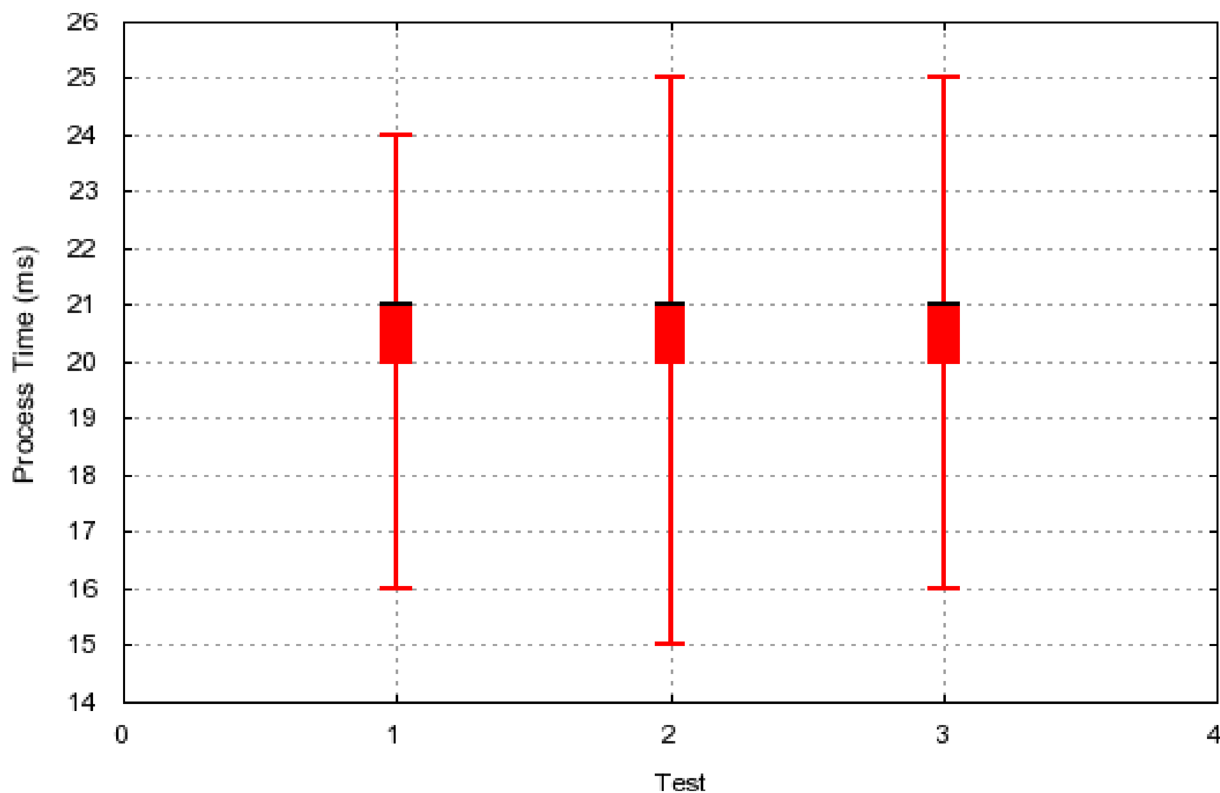


Figura 6.2: Distribuição dos Tempos de Processamento.

Os diagramas de caixa da Figura 6.2 mostram que o tempo de processamento para todos os testes concentra-se entre os valores 20 (vinte) e 21 (vinte e um) milissegundos. Há diferença entre os tempos máximos e mínimos para os, no entanto, para o segundo e terceiro teste, o valor máximo fixou-se em 25 (vinte e cinco) milissegundos. Pode-se considerar que, mesmo com as pequenas diferenças, os tempos de processamento distribuem-se de maneira bastante homogênea para os testes realizados.

Mesmo com metade dos valores observados concentrando-se dentro de um intervalo pequeno, e a distância entre esse intervalo e o maior valor observado ser reduzida, a

escolha do maior valor deixa de penalizar o quarto da amostra maior que o intervalo de maior freqüência.

6.4 Modelos de Previsão

A versão do *proxy* utilizada para medir os tempos de processamento também recupera informações sobre o tráfego de mensagens. Nessas informações constam quantidade e características das mensagens, como tipos (*i.e.* *RT-RPC*, *Promise*, confirmação ou cancelamento), e *deadlines* para cada um dos grupos definidos na seção 5.1.1, e sendo assim, o protótipo implementa 4 (quatro) modelos de previsão no módulo *PS*.

As previsões são realizadas por modelos estatísticos de séries temporais que utilizam informações sobre o volume de tráfego experimentado pelo *proxy* em instantes anteriores, armazenadas no *DB*. Para cada grupo de prioridade é identificado um modelo de série temporal para previsão.

Para determinar os modelos de previsão, as informações sobre o fluxo de mensagens são organizadas em séries para que sejam aplicados os passos definidos na seção 4.2.2, sendo o processo de modelagem de comportamento da rede análogo para cada um dos grupos de prioridade.

Os passos da modelagem envolve o cálculo das funções de auto-correlação e auto-correlação parcial, e a estimativa de parâmetros. Para tais cálculos, foi utilizada a versão 2.4.1 do pacote de ferramentas estatísticas e matemáticas *R* (RProject, 2007).

O *R* implementa o cálculo das funções de auto-correlação e auto-correlação parcial, bem como o método de mínimos quadrados utilizados para estimativa dos parâmetros para cada modelo identificado. A ferramenta também permite gerar comparações entre possíveis modelos de previsão para uma mesma série. Geralmente, essas comparações são feitas com base na variância, no erro médio padrão e no critério de *Akaike* (Box e Jenkins, 1976).

As próximas subseções detalham a identificação do modelo de previsão para cada um dos grupos de prioridade do estudo de caso. Para diminuir o número de modelos implementados no estudo de caso, os grupos de alta e média alta prioridade, assim como os grupos de baixa e média baixa prioridade, são organizados dentro de uma mesma freqüência de requisições e por isso podem utilizar o mesmo modelo de previsão.

6.4.1 Modelo de Previsão Para Mensagens de Média Alta e Alta Prioridade

A arquitetura visa privilegiar as mensagens de prioridade mais alta, principalmente aquelas de requisitos temporais mais rígidos. Para esse tipo de mensagem, a solução ideal seria o redirecionamento imediato, pelo *gateway*, da linha de entrada para a linha de saída correta. Contudo, não é essa a realidade. As mensagens de alta prioridade com requisitos rígidos de tempo disputam de forma igualitária com outras mensagens de menor prioridade.

Mesmo entre mensagens de um mesmo grupo de prioridades ainda existe competição. As mensagens com prioridades iguais podem ser diferentes em *deadline*, por exemplo, e por isso devem ser tratadas de maneiras distintas.

A arquitetura proposta visa manter a ocupação da fila de alta prioridade sempre com a menor ocupação possível, fazendo com que essas mensagens sejam encaminhadas com maior agilidade que as demais. A Figura 6.3 mostra os valores observados para chegadas de mensagens ao *proxy* em diferentes intervalos.

A série descrita pelo gráfico da Figura 6.3 mostra-se estacionária, não apresenta tendência perceptível de crescimento ou decrescimento, e há pouca variação entre as observações de intervalos adjacentes, supondo uma possível dependência entre observações sucessivas. A dependência entre os dados é confirmada pelo cálculo das funções de autocorrelação e autocorrelação parcial da série. As Figuras 6.4 e 6.5 mostram os gráficos das funções de auto-correlação e auto-correlação parcial para a série observada.

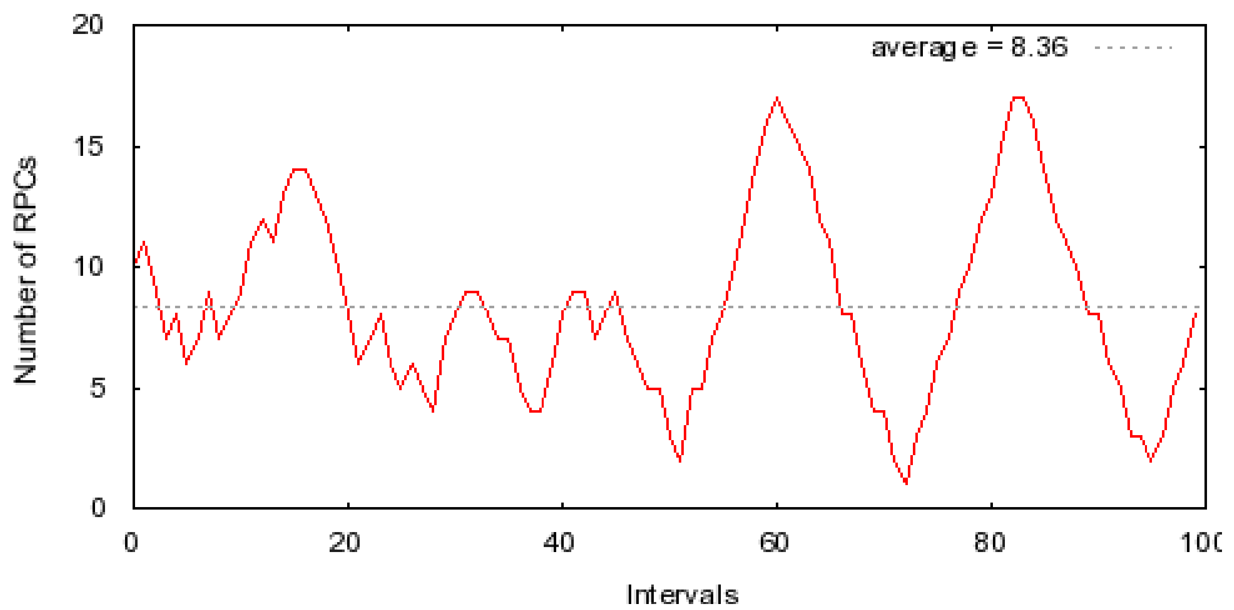


Figura 6.3: Quantidade de RT-RPCs de Alta e Média Alta Prioridade Observada.

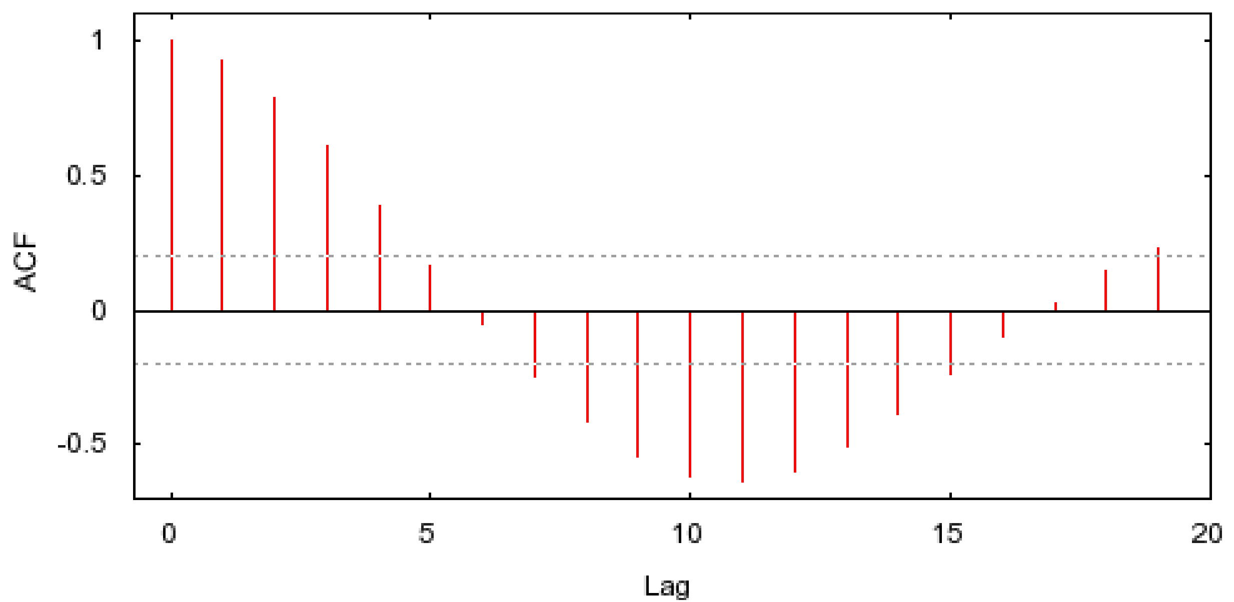


Figura 6.4: Função de Auto-Correlação para RT-RPCs de Alta Prioridade.

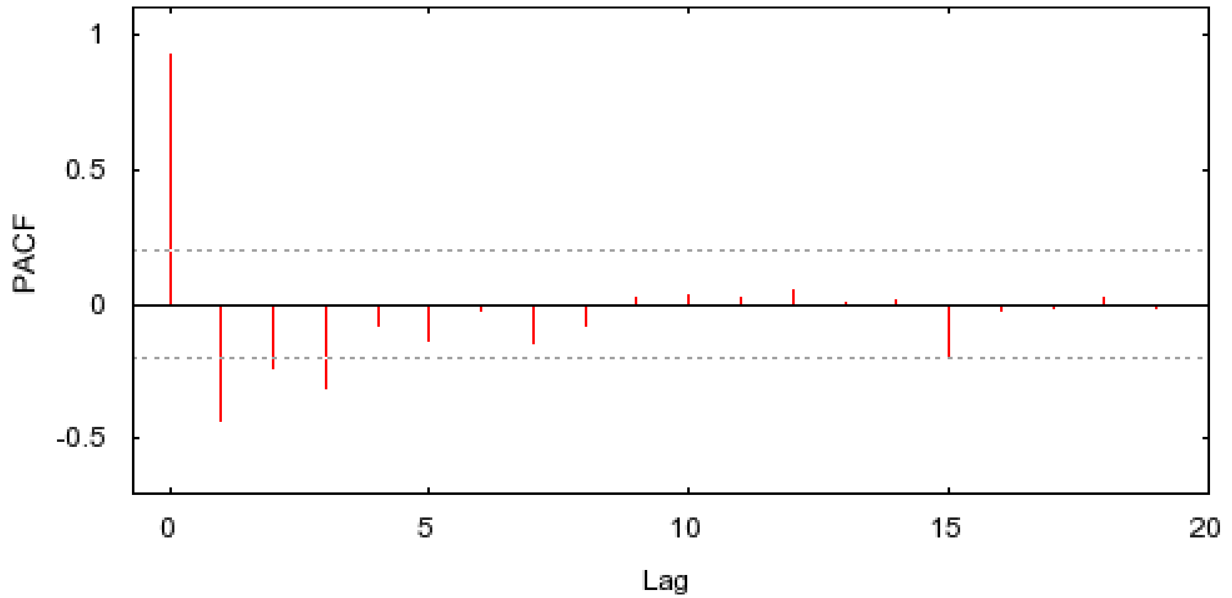


Figura 6.5: Função de Auto-Correlação Parcial para RT-RPCs de Alta Prioridade.

As autocorrelações apresentam uma queda exponencial nos primeiros intervalos seguida por um princípio de senoidais amortecidas, enquanto as autocorrelações parciais apresentam apenas os primeiros valores significativos. Segundo Morettin e Toloi (1981b), uma série com essas características pode ser descrita por um modelo auto-regressivo de quarta ordem, ou simplesmente, *AR(4)* ou *ARIMA(4,0,0)*.

Reescrevendo a Equação 4.20 com os coeficientes estimados para a série, tem-se a Equação 6.1, que permite calcular os valores futuros da série para h intervalos à frente de t.

$$\tilde{Z}_t = 1.1028\tilde{Z}_{t-1} - 0.1017\tilde{Z}_{t-2} + 0.1549\tilde{Z}_{t-3} - 0.3479\tilde{Z}_{t-4} + \epsilon_t$$

$$\tilde{Z}_{t-n} = Z_{t-n} - 8.5575 \qquad N = [0,4];$$

$$Z_t(h) = 1.1028Z_{t-1} - 0.1017Z_{t-2} + 0.1549Z_{t-3} - 0.3479Z_{t-4} + 1.6593 \qquad (6.1)$$

A adequação do modelo foi avaliada por meio da análise dos resíduos do modelo estimado, os quais são considerados ruídos brancos e por isso não devem apresentar dependên-

cia entre si. A função de auto-correlação dos resíduos, ilustrada na Figura 6.6, mostra um passeio aleatório sem valores significativos, independentes e imprevisíveis, garantindo, assim, a adequação do modelo.

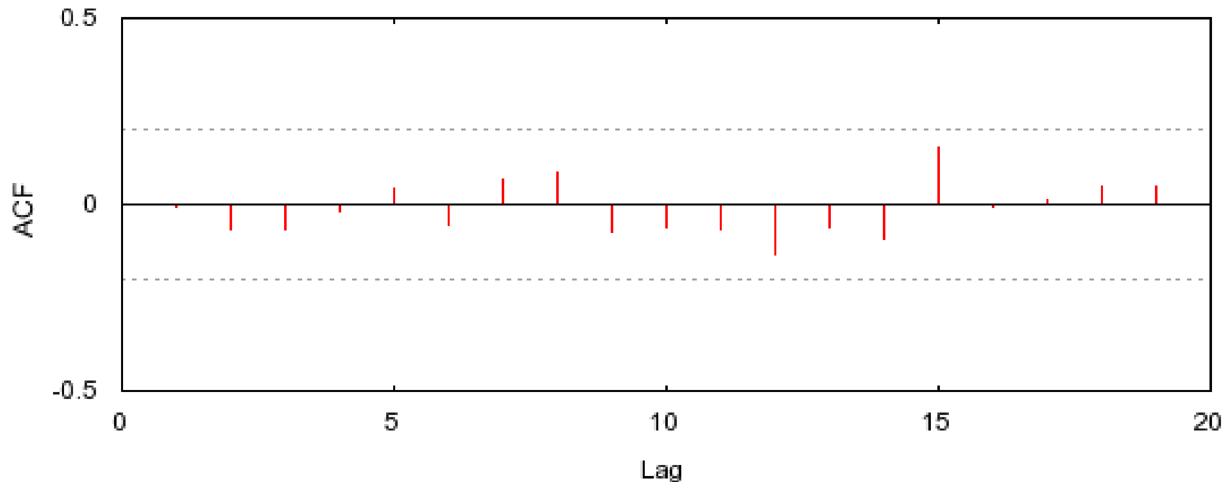


Figura 6.6: Função de Auto-Correlação para os Resíduos do Modelo Estimado AR(4).

Boas previsões só são possíveis com uma modelagem precisa dos valores já observados. Mesmo um modelo adequado pode produzir previsões que não refletem a realidade das observações. Com intuito de adequar o modelo a realidade e minimizar o desvio entre os valores previstos e os valores reais, para cada nova previsão, são desconsideradas as previsões anteriores e considerados apenas os valores reais. A Figura 6.7 ilustra uma comparação entre os valores reais observados e as previsões realizadas pelo protótipo.

Uma vez que o modelo mostra-se adequado, as curvas de valores previstos e observados são bastante próximas e, em certos instantes, coincidentes, permitindo que o proxy utilize das previsões para adequar-se aos estado do ambiente de rede num futuro próximo.

Uma das vantagens da previsão é permitir a alocação antecipada de recursos necessários para atender uma demanda. Para a arquitetura proposta, as previsões fornecem informações que permitem alocar os processadores de fila em número suficiente para atender uma fila de mensagens, minimizando a taxa de falhas de *RT-RPCs* por tempo em fila.

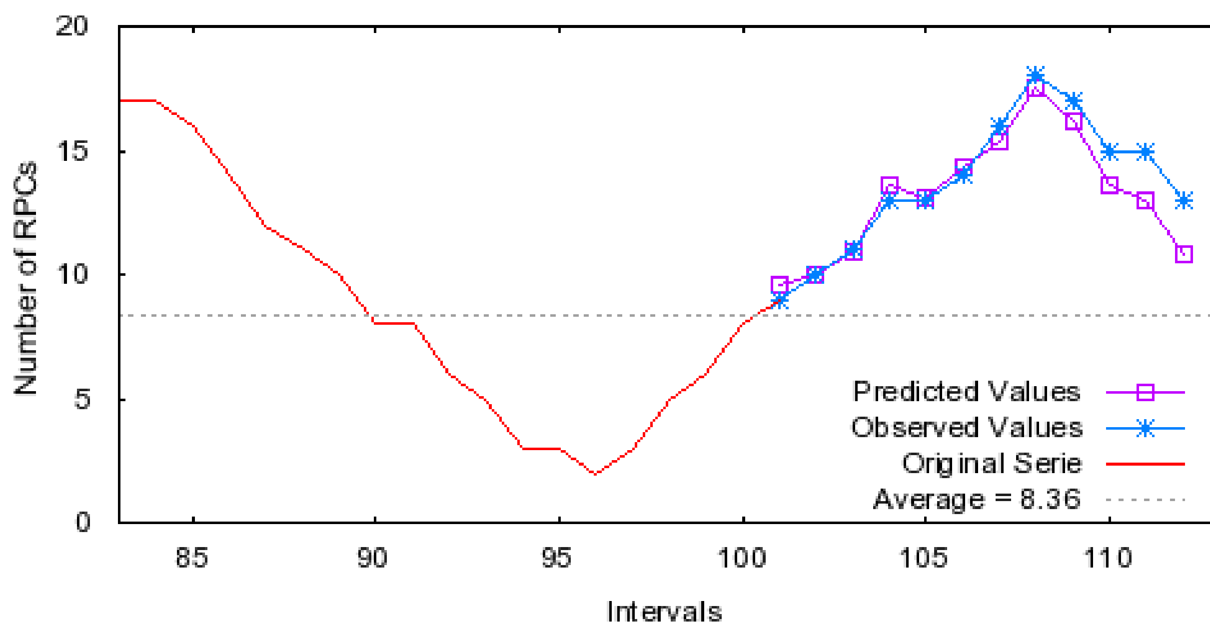


Figura 6.7: Comparação entre Valores Observados e Valores Previstos pelo Modelo AR(4).

6.4.2 Modelo de Previsão Para Mensagens de Baixa e Média Baixa Prioridade

Como o foco principal do estudo de caso é analisar as mensagens de prioridade mais altas, neste estudo de caso o módulo cliente foi configurado para enviar requisições de mensagens de baixa e média baixa prioridade numa mesma freqüência, para diminuir a quantidade de modelagens. A Figura 6.8 ilustra as observações para essas mensagens.

Representando o comportamento normal de uma aplicação que utiliza *RT-RPCs* de prioridade menor na maior parte do tempo, e utiliza chamadas de prioridade maior apenas em situações emergenciais, a freqüência com que *RT-RPCs* de prioridade menor ocorrem é maior como mostrado pela Figura 6.8.

Assim como o número de chegadas observado para *RT-RPCs* de alta prioridade, os valores de chegada para média baixa e baixa prioridade também são descritos por uma série estacionária. As Figuras 6.9 e 6.10 ilustram respectivamente as funções de auto-correlação e auto-correlação parcial.

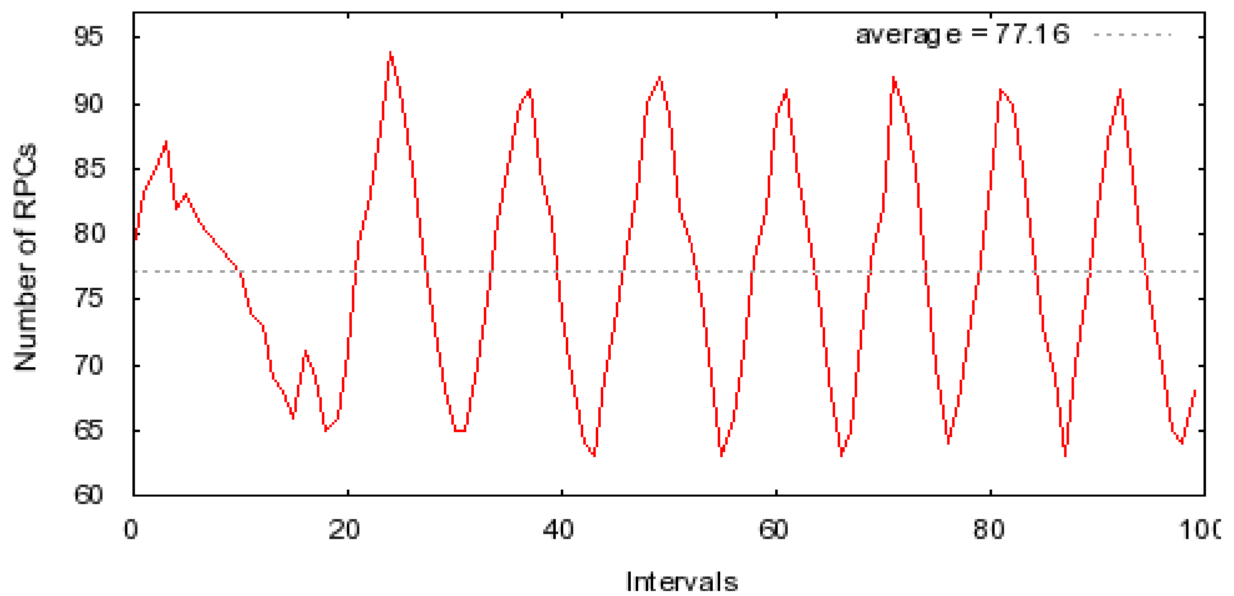


Figura 6.8: Quantidade de RT-RPCs de Média Baixa e Baixa Prioridade Observada.

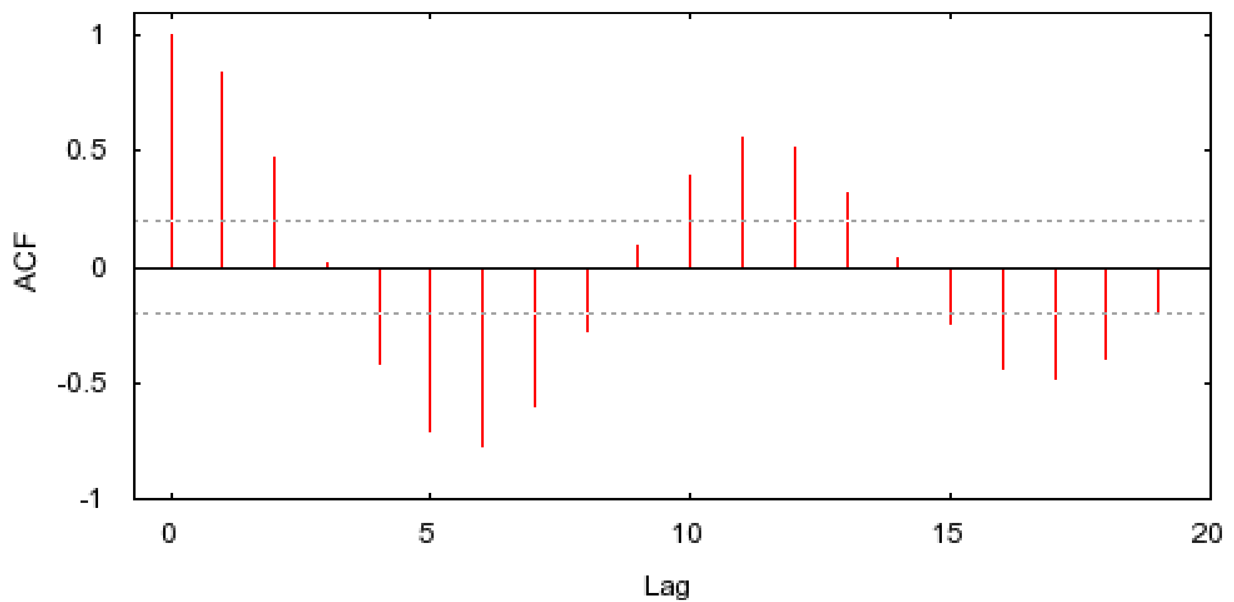


Figura 6.9: Função de Auto-Correlação para RT-RPCs de Média Baixa e Baixa Prioridade.

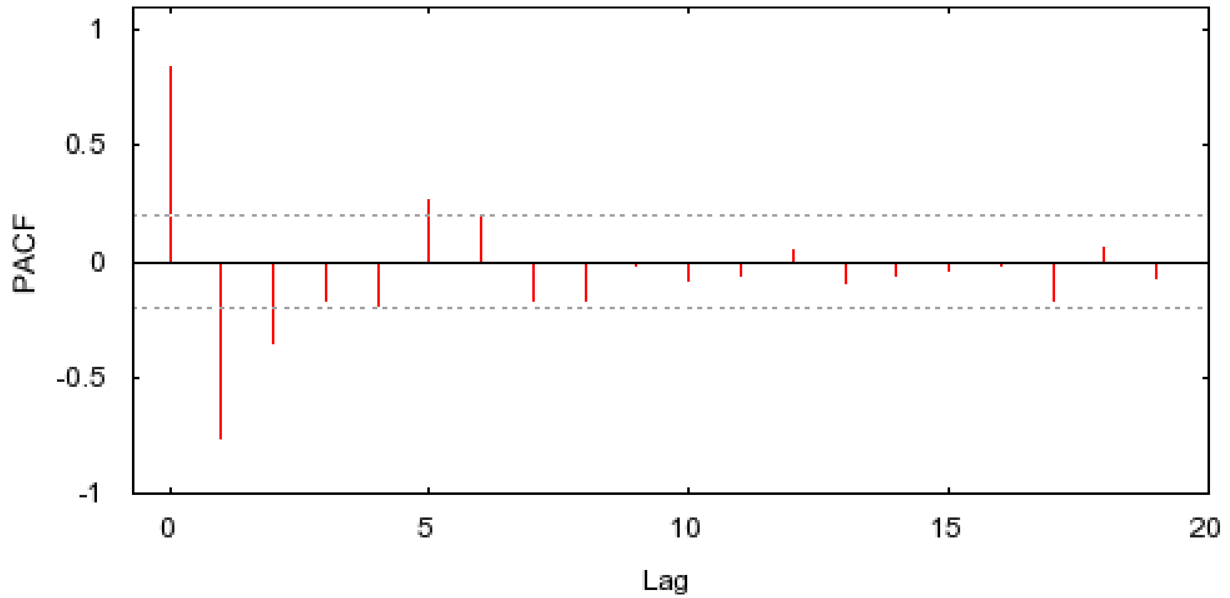


Figura 6.10: Função de Auto-Correlação Parcial para RT-RPCs de Média Baixa e Baixa Prioridade.

As autocorrelações apresentam uma queda exponencial nos primeiros intervalos seguida por de senoidais amortecidas, já as autocorrelações parciais apresentam os 3 (três) primeiros e o sexto valores significativos. Sendo assim, o modelo escolhido para essas observações foi um *AR(3)*.

Reescrevendo a Equação 4.20 com os coeficientes estimados para a série, tem-se a Equação 6.2, que permite calcular os valores futuros da série para h intervalos à frente de t.

$$\tilde{Z}_t = 1.19\tilde{Z}_{t-1} - 0.2037\tilde{Z}_{t-2} - 0.3841\tilde{Z}_{t-3} + \epsilon_t$$
$$\tilde{Z}_{t-n} = Z_{t-n} - 77.1998 \qquad N = [0,3];$$
$$Z_t(h) = 1.19Z_{t-1} - 0.2037Z_{t-2} - 0.3841Z_{t-3} + 30.71 \tag{6.2}$$

A comprovação de adequação do modelo é verificada pela pela aleatoriedade da autocorrelação e pela pouca significância dos resíduos do modelo estimado, como mostrado na Figura 6.11.

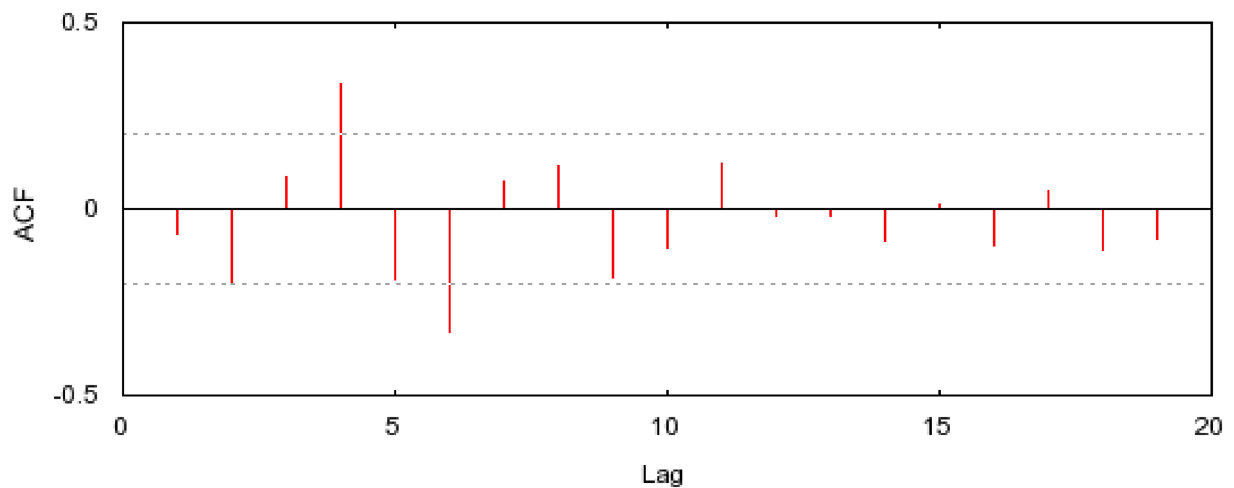


Figura 6.11: Função de Auto-Correlação para os Resíduos do Modelo Estimado AR(3).

Para adaptar-se aos equívocos de previsão, o modelo sempre considera o histórico de valores observados e tenta corrigir as previsões para os próximos momentos. A Figura 6.12 mostra uma comparação entre os valores observados e os valores previstos pelo modelo estimado.

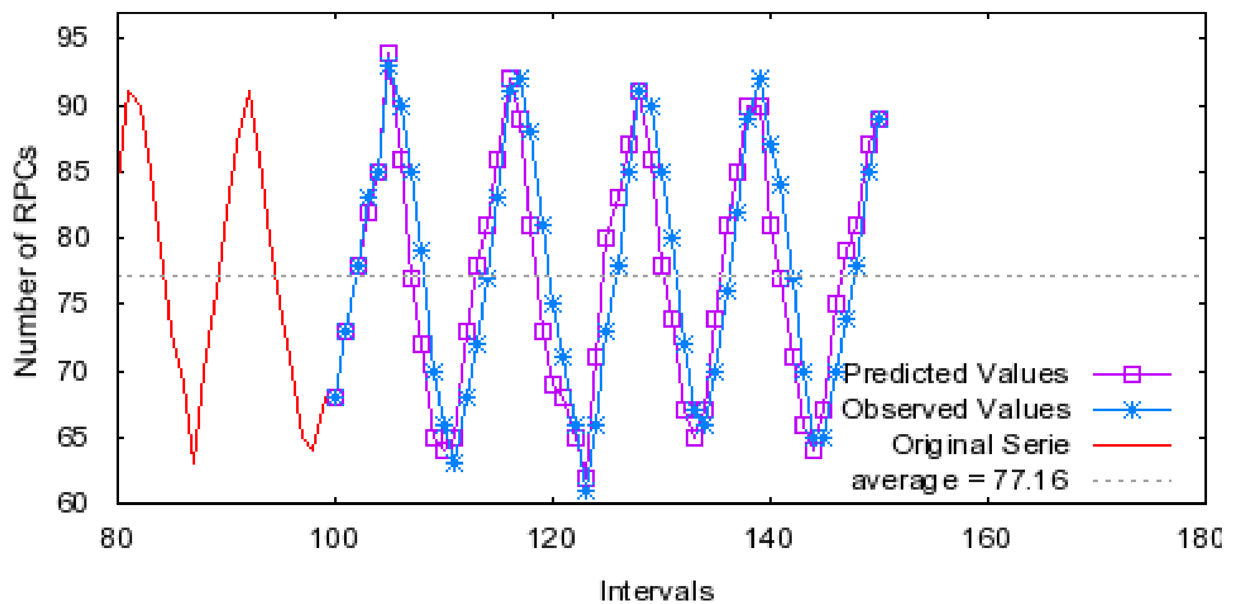


Figura 6.12: Comparação entre Valores Observados e Valores Previstos pelo Modelo AR(3).

A Figura 6.12 mostra que o modelo estimado *AR(3)* consegue obter boas previsões para o tráfego de *RT-RPCs* de média baixa e baixa prioridade. Com isso é possível alocar um número de processadores muito próximo do ideal para esses dois tráfegos.

6.5 Comparação entre Alocação Estática e Dinâmica de Processadores

A comparação entre alocação estática e dinâmica de processadores foi feita com base na quantidade de requisições que falham para cada uma das estratégias de alocação (*i.e.* estática e dinâmica).

O procedimento de testes envolveu a exposição do protótipo há tráfegos idênticos de requisições *RT-RPCs*. Nesses testes, o estados das variáveis e políticas dos módulos da arquitetura são mantidos constantes, exceto, o número de processadores que varia de acordo com a estratégia de alocação utilizada.

Foram realizadas duas abordagens diferentes de testes. Na primeira abordagem, são comparados resultados para a alocação estática de 1 (um) e 5 (cinco) processadores, submetidos a um mesmo fluxo de mensagens. Na segunda abordagem, a estratégia de alocação estática de 5 (cinco) processadores é comparada com a estratégia de alocação dinâmica.

Para cada abordagem, foram realizados testes com o envio de conjuntos de 100 (cem), 250 (duzentas e cinqüenta), 500 (quinhentas) e 1000 (uma mil) requisições *RT-RPCs*. Cada teste analisou a quantidade de requisições canceladas, ou seja, cujos *deadlines* expiraram por espera de atendimento nas filas.

Para realização dos testes de alocação de processadores, a prioridade das mensagens não é uma variável relevante, e por isso foram utilizadas apenas mensagens com prioridade entre 0 (zero) e 7 (sete) (*i.e.* grupo de alta prioridade).

Para minimizar os efeitos do tempo de transmissão e do tempo de execução da chamada no cancelamento das requisições, e, dessa forma, determinar de modo mais confiável qual a influência do tempo de espera por atendimento, nas filas do *proxy*, no sucesso ou falha de uma requisição, foram adicionados atrasos aos processadores de fila do *proxy*. Esses atrasos são iguais para todos os processadores e aumentam a unidade temporal utilizada para medida de *deadline*.

Para não penalizar a execução das requisições com o atraso adicionado aos processadores, o mesmo também foi propagado para o *deadline* das requisições.

Na realização dos testes fixou-se o atraso dos processadores em 50 (cinqüenta) milissegundos, e a política de inserção utilizada para fila de alta prioridade foi por prioridade-*deadline*. A Figura 6.13 mostra o resultado da primeira abordagem de testes comparando a alocação estática de um único processador com a alocação estática de 5 processadores.

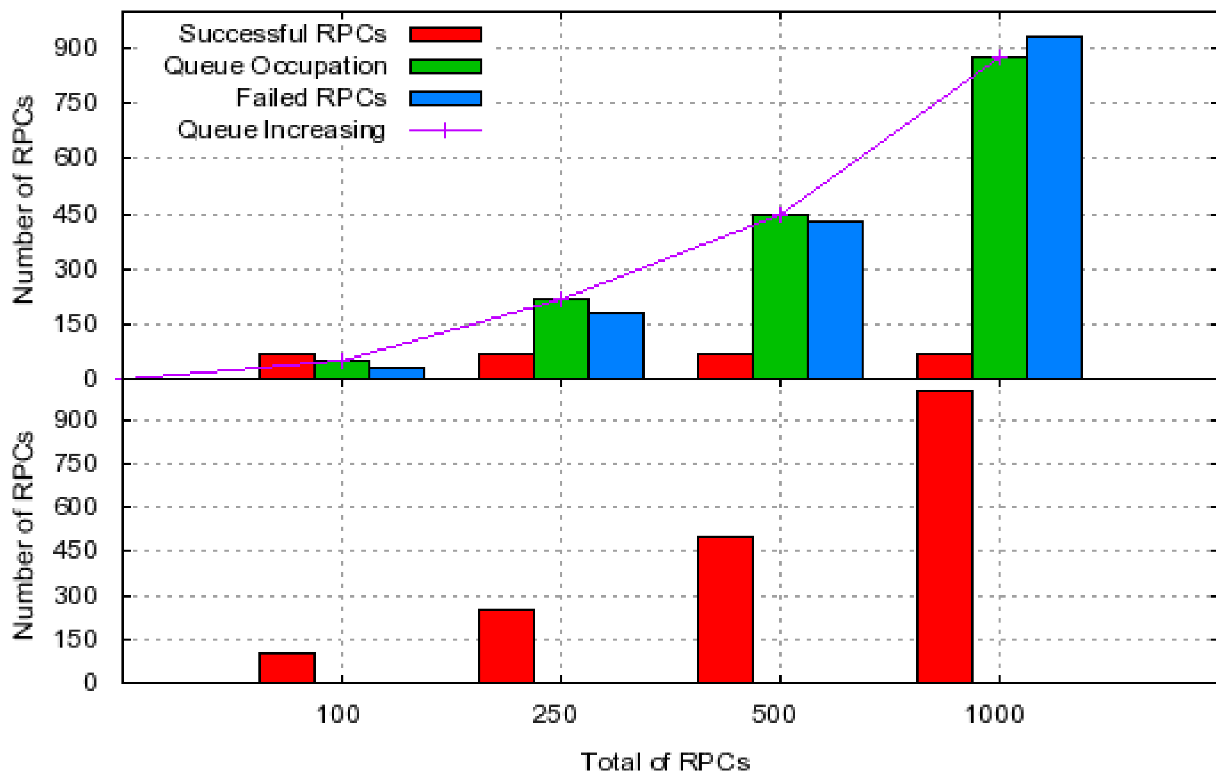


Figura 6.13: Comparação entre Alocações Estáticas de 1 e 5 Processadores.

Pelos histogramas da Figura 6.13 percebe-se que a alocação de um único processador consegue suportar um pequeno número de requisições sem aumento da ocupação de fila. O aumento da ocupação de fila resulta em tempo de espera maior por atendimento, enquanto o processador ocupa-se do processamento de outra requisição. Conseqüentemente, o aumento de fila também resulta em expiramento *deadlines* menores que aguardam por atendimento.

Para quantidade maiores de requisições, o aumento de ocupação da fila e ainda maior, e influencia sobremaneira o número de requisições canceladas, promovendo um efeito de avalanche nas requisições canceladas e provando a ineficiência da alocação estática de um único processador de fila para uma demanda grande de requisições. A Figura 6.14 mostra os resultados para segunda abordagem de testes.

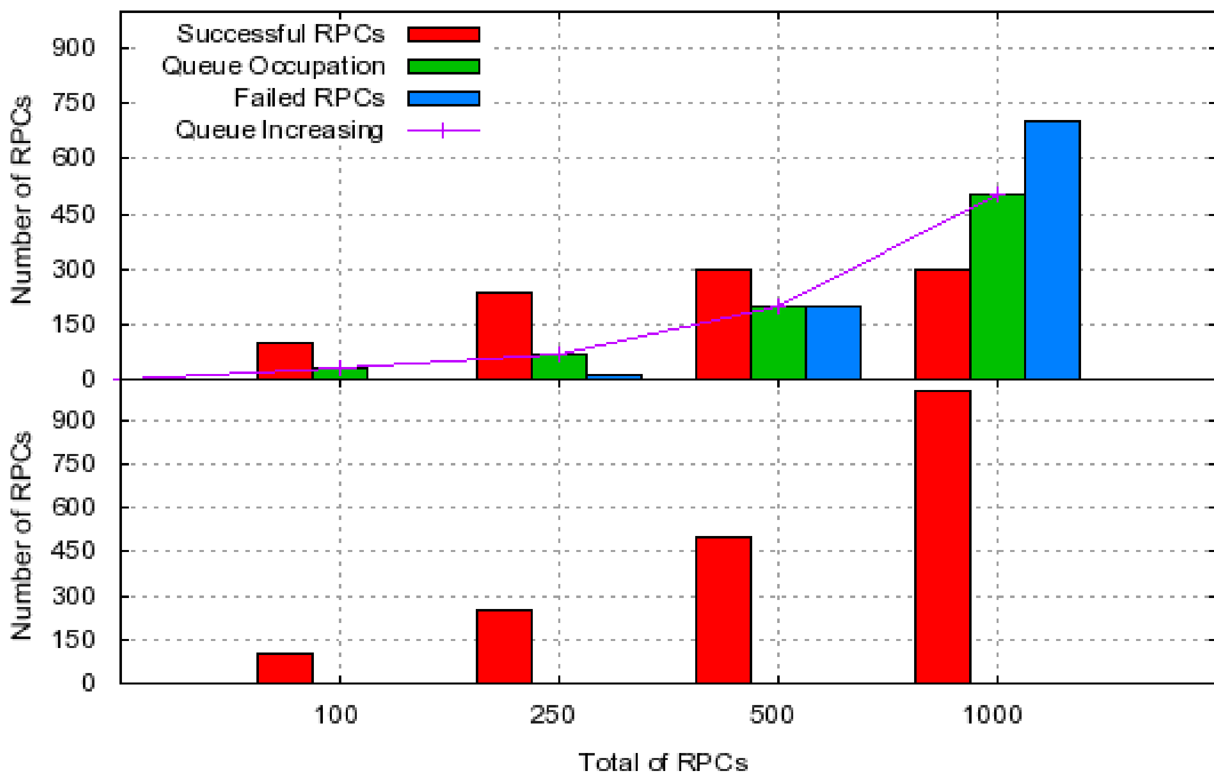


Figura 6.14: Comparação entre Alocação Estática de 5 processadores e Alocação Dinâmica.

A alocação de mais de um processador para uma mesma fila permite que mais de um processador ocupe-se do processamento de uma requisição ao mesmo tempo, e dessa forma, reduza o tempo de espera por atendimento.

Para um número maior de processadores, conclui-se que o número de requisições necessárias até que se perceba um aumento da ocupação da fila é maior que para a alocação estática de um processador. No entanto, para quantidades maiores de requisições, é inevitável a manutenção do tamanho reduzido da fila. Isso se deve ao fato da ocupação de todos os processadores disponíveis para a demanda.

Em contrapartida a alocação estática, foi realizado um teste para alocação dinâmica de processadores. Nesse teste, a alocação é feita com base nas previsões realizadas pelo módulo *PS*.

Os resultados mostram que a ocupação da fila, quando o número de processadores ideal é alocado, permanece sempre reduzido, garantindo um tempo mínimo de espera por atendimento. Dessa forma, o cancelamento eventual de algumas requisições devem-se ao tempo de transmissão e a implementação do servidor, e não podem ser previstos ou resolvidos pela arquitetura.

A alocação estática de processadores de fila é uma alternativa viável quando não há grande demanda de requisições. Contudo, esta estratégia pode não se mostrar eficiente quando do aumento das requisições. A alocação estática também deixa em dúvida quantos processadores devem ser alocados para cada fila, dado que esse número será constante durante toda atuação do *proxy*. Outro aspecto negativo da alocação estática é a inadequada utilização dos recursos disponíveis para o *proxy*, uma vez que, dificilmente, o número de alocados será o ideal para a demanda, sendo sempre maior (*i.e.* sobra de recursos alocados), ou menor (*i.e.* *deficit* de processadores de fila).

6.6 Comparação entre Políticas de Inserção FIFO, por Deadline e por Prioridade-Deadline

Esta seção compara as diferentes políticas de inserção nas filas do módulo *QSM*. As políticas analisadas foram *FIFO* ou *FCFS*, por prioridade e por prioridade-*deadline*. O objetivo é analisar o impacto da política de inserção no êxito ou não de uma requisição.

A comparação entre as diferentes políticas foi feita com base na quantidade de requisições bem sucedidas para cada uma das políticas.

O procedimento de testes envolveu a exposição do protótipo a tráfegos idênticos de requisições *RT-RPCs*. Nesses testes, o número de processadores é mantido constante e apenas a política de inserção empregada pelo *QSM* varia em cada um dos testes.

O número de processadores de fila alocados pelo *QSP* é mantido fora do ideal para que a ocupação da fila não seja nula, garantindo que sempre haverá mensagens aguardando por processamento nas filas do *QSM*. Com isso, é possível analisar quanto a política de inserção e o conseqüente posicionamento de uma mensagem em uma das filas de espera, influencia no sucesso ou não da execução de uma *RT-RPC*.

Para cada política de inserção foram realizados testes com o envio de 1000 (uma mil) requisições *RT-RPCs*, e foram analisados os impactos causados pelas políticas na fila em questão. A Figura 6.15 mostra os resultados obtidos nos testes descritos.

As políticas de inserção diferem em complexidade e custos computacionais. Quanto mais complexa a política, maior o custo de uma inserção. No entanto, é necessário estabelecer um balanceamento entre o custo de inserção e o custo de posicionamento no cumprimento das *deadlines*. A comparação efetiva dessas políticas permite limitar os cenários nos quais estas devem ser empregadas, baseando-se no benefício de posicionar corretamente uma requisição em relação ao custo desprendido pela inserção. Com isso, é possível determinar em que momentos o *DeM* deve alternar entre as diferentes políticas de inserção.

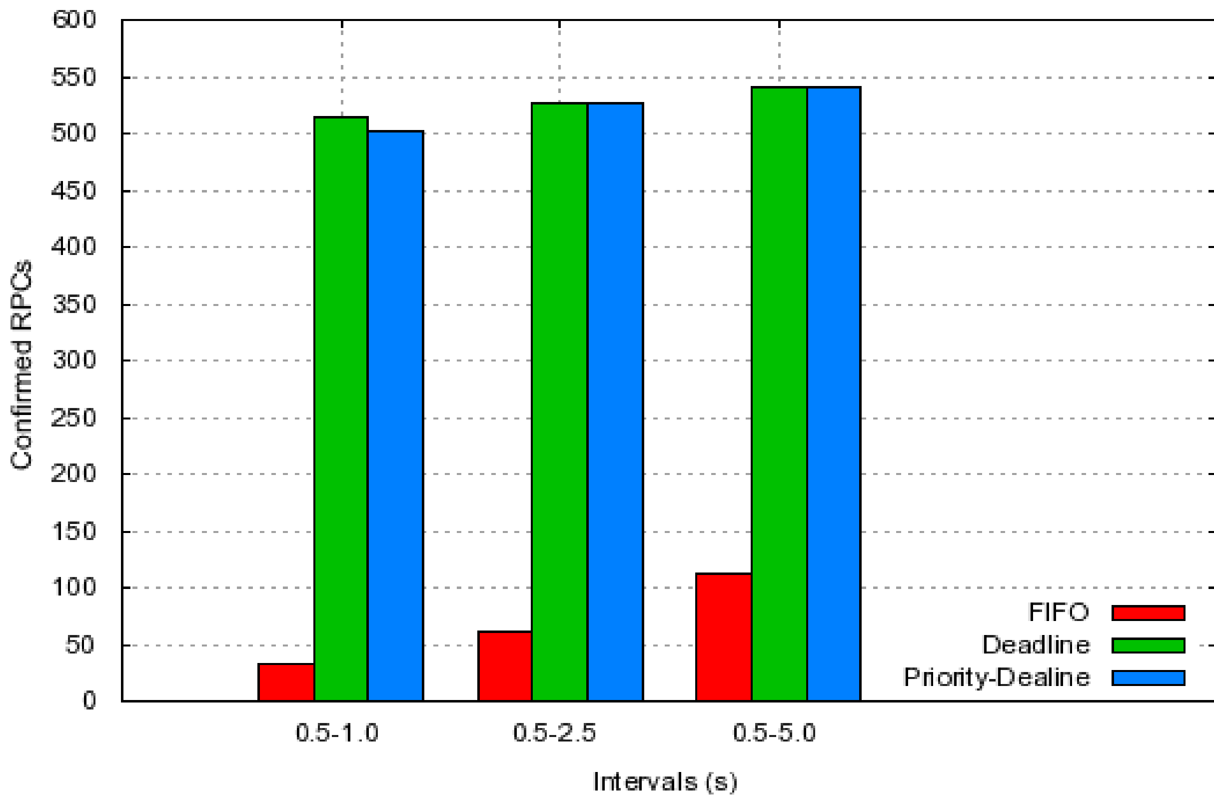


Figura 6.15: Comparação entre Políticas de Inserção.

Os resultados mostram que apesar da política *FIFO* ter complexidade *O(1)*, ou seja, inserção direta e sem busca, esta apresenta os piores resultados independente do intervalo de *deadlines*. Esse resultado permite concluir que a economia de tempo para inserção não é suficiente para garantir o cumprimento dos requisitos temporais.

Também é possível observar que quando o *deadline* atinge valores maiores como para o segundo e terceiro intervalos de testes, as inserções por prioridade e prioridade-*deadline* apresentam resultados idênticos, mostrando que a sobrecarga de buscar o grupo de prioridades na inserção é irrelevante quando os requisitos de tempo são menos rígidos.

Mesmo com melhores políticas de inserção, parte das requisições ainda falham, isso deve-se principalmente a alocação estática e inadequada dos processadores de fila utilizada neste teste.

Com os testes realizados é possível concluir que o tempo de inserção de uma requisição nas filas do *proxy* não é tão relevante no cumprimento dos requisitos temporais quanto

a posição que a requisição ocupa na fila. Sendo assim, apesar de políticas de menor esforço garantirem uma inserção mais rápida, não garantem um bom posicionamento da requisição e por isso apresentam resultados inferiores quando comparadas às políticas de maior esforço.

Capítulo 7

Conclusão

Os testes realizados sobre o protótipo validam a arquitetura proposta. Os resultados mostram que a arquitetura consegue atingir seu objetivo de privilegiar as mensagens de prioridades mais altas, principalmente aquelas de requisitos temporais mais rígidos.

Para um dado ambiente de rede conhecido, de aplicações específicas que utilizam *RT-RPCs*, e, desde que esse ambiente não sofra variações significativas no tráfego com relação aos dados históricos observados, os modelos de previsão baseados no método de *Box-Jenkins* conseguem realizar boas previsões sobre o volume e características do tráfego para instantes brevemente futuros. Dessa forma, a arquitetura pode se ajustar antecipadamente, com bastante precisão, aos estados futuros do ambiente de rede.

As previsões permitem alocar dinamicamente o número de processadores mais próximo do ideal para atender uma demanda de requisições *RT-RPCs*. A alocação dinâmica de processadores visa a melhor utilização dos recursos disponíveis no *proxy*. O estudo de caso mostrou que a alocação estática de processadores não suporta o aumento na demanda de requisições, resultando num aumento de ocupação da fila. Mesmo quando um grande número de processadores é alocado estaticamente sob uma demanda é pequena, muitos desses processadores ficam ociosos grande parte do tempo, superutilizando desnecessariamente os recursos do *proxy* de maneira inadequada.

A alocação dinâmica de processadores com base nas previsões, otimiza a utilização dos recursos e busca manter a ocupação da fila sempre a menor possível. A alocação dinâmica de processadores garante também que sempre a maior parte dos processadores estão ocupados, fazendo sempre boa utilização dos recursos alocados.

A base de políticas permite que o *proxy* seja auto-ajustável, optando sempre pela política que melhor se adapta ao estado atual de tráfego. Usando de políticas, o *proxy* pode adaptar os recursos utilizados aos recursos necessários, optando ora por algoritmos de menor complexidade, menos eficientes, mas suficientes para um determinado estado. Este trabalho não explorou as políticas com rigor e considerações são feitas na seção 7.2.

Os resultados mostram também que mesmo com todas as otimizações implementadas no protótipo, parte das mensagens ainda é perdida. Esse fato deve-se principalmente a requisitos de tempo extremamente rígidos ou então a implementação dos servidores, que fogem do escopo deste trabalhos.

7.1 Expectativa de Uso da Arquitetura

Espera-se que com a conclusão e maturidade da arquitetura, a mesma seja utilizada para diferenciar o tráfego de *RPCs* de tempo real com base nas características de prioridade e *deadline* das chamadas.

Com isso, diversas áreas de pesquisa, não só da computação, poderiam utilizá-la para garantir o tratamento adequado a *RPCs* de maior prioridade quando necessária a diferenciação num ambiente de rede compartilhado.

Num primeiro momento, espera-se que a arquitetura seja utilizada para controle remoto de experimentos laboratoriais que necessitem de *feedback* do responsável em horários não regulares, como por exemplo, experimentos de *Skinner box* e acompanhamento de processos biológicos, como culturas celulares, além do controle remoto de processos químicos.

Com aperfeiçoamento da arquitetura há ainda a expectativa de que a mesma forneça meios confiáveis para sua utilização em campos científicos de aplicações mais rígidas como telemedicina e educação à distância.

7.2 Trabalhos Futuros

Os trabalhos futuros envolvem a adaptação do *DeM* para tornar a base de políticas mais dinâmicas. Atualmente as políticas são parametrizadas e configuráveis, mas ainda assim limitadas. Uma das modificações a serem implementadas é capacidade de atualização dinâmica da *PDB*, sem que seja necessária a modificação de código do *proxy*.

Outro aspecto relacionado as políticas está em definir uma abordagem para especificação de políticas para o *QSP*. Essas políticas visam controlar a prioridade e o escalonamento dos processadores de maneira adequada de acordo com a fila para qual foram alocados. Essa abordagem também deve permitir que processadores alocados para filas menores prioridades sejam realocados de maneira inteligente para filas de maiores prioridades quando na ausência de processadores livres.

Outros trabalhos incluem a adaptação da arquitetura para que sejam considerados perfis de dispositivos móveis (*e.g. smartphones* e *PDAs*) na tomada de decisão. Uma vez que dispositivos móveis possuem recursos bastante limitados quando comparados a *desktops* ou *laptops*, é necessário que essas limitações sejam levadas em consideração em aplicações que envolvam *RT-RPCs* para que tais dispositivos concorram de forma igualitária com aqueles de melhores recursos.

Referências Bibliográficas

The common object request broker: Architecture and specification, revision 2.3.1. Omg document, Object Management Group, 1999.

Introduction to quality of services (qos). White paper, Nortel Networks, http://www.nortel.com/products/02/bstk/switches/bps/collateral/56058.25_-022403.pdf, 1999.

The need for qos. White paper., Stardust Technologies, http://www.qosforum.com/white-papers/Need_for_QoS-v4.pdf, 1999.

Bluetooth specification version 2.0 + edr. Bluetooth SIG, 2004.

Java idl. 2006.
Disponível em: <`http://java.sun.com/products/jdk/idl/`>. Acesso em: 30/01/2006

Aimoto, T.; Miyake, S. *Overview of DiffServ technology: its mechanism and implementation.* 2000.

Amigo Amigo: Ambient intelligence for the networked home environment. 2004. Disponível em: <`http://www.hitech-projects.com/euprojects/amigo/`>

Black, D.; Blake, S.; Carlson, M.; Davies, E.; Wang, Z.; Weiss, W. *An Architecture for Differentiated Services.* 1998.

Box, G. E. P.; Jenkins, G. M. *Time series analysis.* Holden Day, 1976.

BRADEN, R. *Zhang, L., Berson, S., Herzog, S. and S. Jamin,"Resource ReSerVation Protocol (RSVP)–Version 1 Functional Specification.* 1997.

CROW, B.; WIDJAJA, I.; KIM, L.; SAKAI, P. *IEEE 802.11 Wireless Local Area Networks.* 1997.

DEITEL, H. M.; DEITEL, P. J. *Java, como programar.* 3. ed. Porto Alegre: Bookman, 2001.

DIAZ, D. Gnu prolog. 2007.
Disponível em: <http://www.gprolog.org/>

FERGUSON, P.; HUSTON, G. *Quality of service: Delivering qos on the internet and in corporate networks.* Wiley Computer Publishing, 1999.

GANZ, A.; WONGTHAVARAWAT, K.; PHONPHOEM, A. Q-soft: software framework for qos support in home networks. *Computer Networks*, v. 42, p. 7–22, 2003.

GRITZALIS, S.; ILIADIS, J.; OIKONOMOPOULOS, S. Distributed component software security issues on deploying a secure electronic marketplace. *Information Management & Computer Security*, v. 8, n. 1, p. 5–13, 2000.

HWANG, W.; TSENG, P. *A QoS-aware residential gateway with bandwidth management.* 2005.

JOHNSON, L.; MONTGOMERY, D. C. *Operations research in production planning, scheduling, and inventory control.* John Wiley & Sons, 1974.

KENDALL, D. G. Some problems in the theory of queues. *Journal of the Royal Statistical Society. Series B (Methodological)*, v. 13, n. 2, p. 151–185, 1951.

KIM, D.-S.; LEE, J.-M.; KWON, W. H.; YUH, I. K. Design and implementation of home network systems using upnp middleware for networked appliances. *Consumer Eletronics*, v. 48, p. 963– 972, 2002.

KUMMEL, S.; HUTSCHENREUTHER, T.; SCHILL, A. *QoS support for advanced RPC interactions.* 1997.

LARSON, R. C.; ODONI, A. R. *Urban operations research.* New Jersey: Prentice-Hall, 1981.

LEE, J. Design of a communication system capable of supporting real-time rpc. *iceccs*, v. 00, p. 99, 1996.

LEI, B.; ANANDA, A.; TECK, T. *QoS-aware residential gateway.* 2002.

LITTLE, J. D. C. A proof for the queuing formula: L= lambda w. *Operations Research*, v. 9, n. 3, p. 383–387, 1961.

MAISTER, D. H. Note on the management of queues. *Harvard Business Review*, p. 1–14, 1995.

MELO, E. T. L. *Qualidade de serviço em redes ip com diffserv: Avaliação através de medições.* Dissertação de Mestrado, Universidade Federal de Santa Catarina, 2001.

MORETTIN, P. A.; TOLOI, C. M. D. C. *Modelos para previsão de séries temporais*, v. 1. Instituto de matemática pura e aplicada, 1981a.

MORETTIN, P. A.; TOLOI, C. M. D. C. *Previsão de séries temporais.* Atual Editora, 1981b.

OMG *Object management group.* 2006.
Disponível em: <`www.omg.org`>. Acesso em: 29/01/2006

PARK, K. I. *Qos in packets networks.* Boston: Springer Science, 2005.

POSTEL, J. *RFC 791: Internet Protocol.* 1981.

REDMOND, F. E. *Dcom: Microsoft distributed component object model.* IDG Books Worldwide, 1997.

RFC2205 Resource reservation protocol (rsvp) version 1 functional specification. 1997.

RPROJECT *The r project for statistical computing.* 2007. Disponível em: <http://www.r-project.org/>

RUBIN, W.; BRAIN, M. *Understanding dcom.* Upper Saddle River: Prentice Hall, 1999.

SATYANARAYANAN, M. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, v. 8, n. 4, 2001.

STONE, C. M. *Object management architecture guide.* 3. ed. John Wiley & Sons, 1995.

TANENBAUM, A. S.; STEEN, M. V. *Distributed systems: Principles and paradigms.* Englewood Cliffs: Prentice Hall, 2002.

TEITLEBAUM, B.; HANS, T. *Qos requirements for internet2.* Relatório técnico internet2, http://www.internet2.edu/qos/may98Workshop/html/requirements.html, 1998.

VILLELA, R. T. N. *Gerador de interfaces gráficas com suporte à chamada remota de procedimentos de tempo real.* Dissertação de Mestrado, UFSCar, São Carlos, 2001.

VINOSKI, S. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, v. 35, n. 2, 1997.

WANG, Q.; YE, Q.; CHENG, L. *An Inter-application and Inter-client Priority-based QoS Proxy Architecture for Heterogeneous Networks.* 2005.

WERNER, L.; RIBEIRO, J. L. D. Previsão de demanda: Uma aplicação dos modelos de box-jenkins na área de assitência técnica de computadores pessoais. *Gestão & Produção*, v. 10, n. 1, p. 47–67, 2003.

WILLLIAMS, S.; KINDEL, C. *The component object model: A technical overview.* Relatório Técnico, Microsoft Corporation, 1994.
Disponível em: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncomg/html/msdn_comppr.asp>

ZHANG, L.; DEERING, S.; ESTRIN, D.; SHENKER, S.; ZAPPALA, D. *RSVP: a new resource reservation protocol.* 2002.