UNIVERSIDADE FEDERAL DE SÃO CARLOS
CENTRO DE CIÊNCIAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO
DEPARTAMENTO DE COMPUTAÇÃO

Integração dos Requisitos Temporais de um Kernel de Tempo-Real e de sua Comunicação em Redes

Aluno: Daniel Augusto Rossler
Área: Sistemas Distribuídos e Redes
Orientador: Prof. Dr. Célio Estevan Moron

São Carlos
2004

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

R836ir

Rossler, Daniel Augusto.
Integração dos requisitos temporais de um Kernel de tempo-real e de sua comunicação em redes / Daniel Augusto Rossler. -- São Carlos : UFSCar, 2004.
153 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2004.

1. Sistemas operacionais distribuídos (computadores). 2. Tempo real. 3. Rede de computadores. I. Título.

CDD: 005.44 (20ª)

Agradecimentos:

Aos meus pais e namorada, pelo incentivo e apoio durante todos os meus estudos.

Ao Prof. Dr. Célio Estevan Moron, pela orientação, dedicação e paciência dispensada na realização deste trabalho.

A todos docentes do Departamento de Computação que participaram de minha formação acadêmica.

Agradeço também a todos os amigos e colegas que contribuíram com sugestões ao meu trabalho.

Lista de Figuras

Sumário

Resumo

O estudo realizado neste trabalho tem por foco estender a troca de mensagens entre processos em um kernel de tempo-real, que atualmente é executada entre processos de uma mesma estação, de modo a abranger também processos em estações distintas.

De modo a garantir a transparência de localização dos processos utilizaremos um servidor de nomes para gerenciar os processos do sistema.

E para melhorar o desempenho de acesso a este servidor, as técnicas de cache e prefetching serão empregadas.

Também, durante a transmissão de uma mensagem pelo sistema, o protocolo de Priority Ceiling é utilizado para definir a prioridade do processo receptor da mensagem.

Como resultado deste trabalho, foi desenvolvido um Sistema Operacional de Tempo Real com as características descritas acima, e com o nome de SãoCarlOS.

Abstract

The study performed on this paper is intended to extend the message exchange between process of a real time kernel, that nowadays is executed between processes on the same station, in order to cover also distinguish stations.

In order to allow location transparency to the processes, a name server is used to manage the system processes.

And to optimize the server access, the cache and pre-fetching techniques are utilized.

Also, during the arrival of a new message to a system node, the Priority Ceiling Protocol will be used to define the priority of the process that will receive the message.

As result of this work, it was developed a Real Time Operating System with all the features described above. The name of this Operating System is SaoCarlOS.

Capítulo 1
Introdução

Atualmente assistimos a uma grande expansão dos sistemas de tempo-real e sistemas embutidos. Como fronteira do desenvolvimento temos a conexão destes sistemas e dispositivos com a Internet. Historicamente tivemos o desenvolvimento dos sistemas operacionais de tempo real, ou RTOS (*Real-Time Operating System*), totalmente desassociados dos subsistemas de comunicação em redes. Sendo assim, hoje as abordagens são diferenciadas para o tratamento dos requisitos temporais nos RTOS e nos subsistemas de comunicação.

A integração desses dois aspectos dos sistemas de tempo-real tornará possível fazer a comunicação entre processos em máquinas distintas, controlar suas estruturas de sincronismo usando semáforos, filas, exclusão mútua e interrupções, levando em conta os requisitos temporais.

1.1 Abordagem sobre Troca de Mensagens com Requisitos Temporais

Algumas aplicações possuem troca de mensagens com restrições de tempo. Algumas mensagens necessitam que sua entrega ocorra antes de um determinado prazo. Outras mensagens precisam ter garantias que vão chegar antes de uma determinada mensagem. Também existem mensagens periódicas que ocorrem a cada intervalo fixo de tempo e necessitam que seu tratamento ocorra com a mesma periodicidade.

Com o intuito de poder prover garantias temporais durante a manipulação destas mensagens, são atribuídas prioridades a um cabeçalho destas mensagens. Tais prioridades nas mensagens afetam o escalonamento de processos do sistema, permitindo que um processo receba uma mensagem de alta prioridade, como descrito por [2]. Também a escolha da próxima mensagem a ser transmitida pela rede de comunicação é feita com base na criticalidade da mensagem como descrito por [3].

Esta nova política de atendimento para troca de mensagens será baseada em um Middleware de Tempo Real e em chamadas de sistema do kernel SãoCarlOS. Por meio de suas interfaces de programação, será possível que desenvolvedores da Camada de Aplicação tenham maior flexibilidade nas aplicações de tempo-real. Tal flexibilidade disponibiliza à aplicação de tempo real, que utiliza as Chamadas de Sistema (ou System Calls) de um RTOS, escolha os parâmetros mais adequados para a política de transmissão de suas mensagens.

1.2 Segmentação dos Capítulos

Este texto foi dividido de modo a agrupar as subseções em 4 grandes tópicos: Tempo Real, Sistemas Operacionais, Rede de Comunicação e Assuntos Relacionados ao Projeto.

No Capítulo 2 será tratada a Classificação de STRs (Sistemas de Tempo Real) e de Sistemas Embutidos. Mais ao final do capítulo, serão levantados alguns parâmetros de tempo real que podem ser utilizados para definir os requisitos temporais de uma mensagem.

Já no Capítulo 3 constarão algumas estruturas e funcionalidades de um Sistema Operacional para contextualização do leitor para os capítulos seguintes. Neste capítulo será dada ênfase para Sistemas Operacionais de Tempo Real.

O Capitulo 4 é responsável por mostrar as principais características de uma rede de comunicação, assim como definir os parâmetros necessários para a política de atendimento durante a troca das mensagens.

No Capítulo 5 serão abordados alguns assuntos referentes à Computação Distribuída. Também serão vistos neste capítulo alguns exemplos de Registro de Aplicações remotas, como CORBA, CORBA RT, RMI e DCOM. Este capítulo dará o embasamento teórico para o servidor de nomes definido no Capítulo 6.

No Capítulo 6 será exibida a base para o desenvolvimento do projeto e demais conclusões e considerações.

O Capítulo 7 foi reservado para descrever as funcionalidades e características da implementação das chamadas de sistema do Kernel SãoCarlOS.

No Capítulo 8 são feitos testes baseados em trocas de mensagens e análises estatísticas sobre os tempos destas trocas de mensagens.

O Capítulo 9 contem a conclusão do trabalho, assim como a descrição dos trabalhos futuros.

Capítulo 2
Tempo-Real

Um sistema que suporta uma aplicação de tempo real deve garantir o cumprimento das garantias temporais daquela determinada aplicação. Atender tais garantias implica em obter uma resposta correta dentro de um intervalo de tempo pré-definido. Tradicionalmente, são atribuídos valores a um processo indicando o quão crítico este processo é para o sistema, de forma que tal processo possa ser executado antes de processos menos críticos, e assim cumprir seu requisito temporal. Um exemplo de um RTOS que trabalha desta maneira é o Virtuoso (tm) [31].

Neste capítulo serão analisados alguns parâmetros de tempo real que podem ser transmitidos acoplados a uma mensagem de modo a prover garantias temporais a esta transmissão. Tais parâmetros serão base para todo o estudo deste trabalho.

2.1 Sistemas de Tempo Real

Sistemas de Tempo Real normalmente estão presentes em controle de processos, em refinarias, no controle de vôo e em alguns projetos militares. De um modo geral, eles são aplicados para prover funcionalidades de Tempo Real às Aplicações de Tempo-Real.

Segundo [13], Aplicações de Tempo-Real impõem requisitos temporais rígidos ao comportamento do Sistema. Sendo assim, o tipo de sistema que suporta e garante que tais requisitos serão cumpridos durante a execução de Aplicações de Tempo-Real é referenciado como Sistemas de Tempo Real (ou STR).

Em um Sistema de Tempo Real, tanto a resposta correta quanto o cumprimento dos requisitos temporais são importantes para o funcionamento do Sistema. Portanto, em um STR, mesmo que a resposta esteja correta, mas os requisitos temporais não forem cumpridos, o sistema terá falhado.

2.2 Classificação de Sistemas de Tempo-Real

Como descrito em [13] é possível classificar Sistemas de Tempo-Real como Críticos (ou *Hard Real-Time*) e não-Críticos (ou *Soft Real-Time*).

Sistemas de Tempo Real Críticos são aqueles cujo não cumprimento de qualquer tipo requisito temporal causará uma falha no sistema, pois geralmente estes sistemas estão encarregados de administrar situações críticas, como aparelhos hospitalares responsáveis pela manutenção de vidas humanas ou controle de catástrofes ambientais.

Sistemas de Tempo Real não-Críticos são aqueles cujo não comprimento ocasional de um requisito temporal não compromete o sistema. Por exemplo, um telefone pode perder uma chamada em um milhão, caso esteja com as linhas sobrecarregadas, e ainda estará dentro de sua especificação.

Na prática ainda existem alguns sistemas intermediários em que perder um prazo pode fazer com que seja necessário parar uma atividade que está ocorrendo e reiniciá-la, mas não é nada fatal.

2.3 Sistemas Embarcados

Uma das mais freqüentes utilizações de um sistema de tempo real é em um Sistema Embarcado. De acordo com [13] quando um determinado sistema computacional é utilizado por um sistema maior para prover controle e funções computacionais, ele é referido como um Sistema Embarcado. Normalmente este tipo de sistema utiliza hardware dedicado ou controladores para controlar muitos sistemas.

Para controlar e gerenciar o sistema, os Sistemas Embarcados (*Embeded Systems*) coletam de dados através de sensores e interagem com outros componentes eletrônicos.

2.4 Concorrência

A concorrência em sistemas de Tempo-Real é dada através de Tarefas (*Tasks*) Concorrentes e Interrupções. A criação dessas tarefas pode ser Estática, quando as tarefas são determinadas em tempo de projeto, ou Dinâmicas onde há a possibilidade de se criar novas tarefas em tempo de execução.

Uma grande vantagem de utilizar-se de criação dinâmica de tarefas é a flexibilidade de programação. Porém esta flexibilidade tem um custo: a dificuldade de manter a previsibilidade (ou *predictability*) durante a alocação de recursos.

2.5 Parâmetros de Tempo-Real

Durante a política de atendimento das mensagens originadas de tarefas de tempo-real alguns parâmetros são analisados. Tais parâmetros serão empregados nos capítulos posteriores e estarão relacionados com o escalonador de processos e a política de atendimento de mensagens do sistema.

Um dos parâmetros mais importantes de sistemas de tempo real é o Prazo (ou *Deadline*). O Prazo é o valor de tempo máximo até o qual a execução de uma tarefa deve ser concluída. Em um STR, o não cumprimento de um Prazo implica em uma falha no sistema.

Um processo também pode manter os parâmetros de Tempo de Início (*Start Time* ou 's') e Tempo de Término (*Finishing Time* ou 'f'), que estão relacionadas respectivamente com a criação do processo e a conclusão de sua execução. Através do parâmetro de Término é possível calcular o Folga (*Laxity / Slack*) de uma tarefa em relação ao seu Prazo (*Laxity* = *Deadline* - f).

Também utilizado em STRs é a Prioridade (*Priority*). O uso da Prioridade permite denominar uma escala de valores inteiros para tarefas mais prioritárias e para as menos prioritárias.

Um outro parâmetro utilizado em STRs é a Criticalidade (Criticality). Através da Criticalidade é possível denominar valores inteiros para Prazos não-Criticos (*Soft Deadlines*) e para Prazos Críticos (*Hard Deadlines*).

A partir do não cumprimento de um Prazo Crítico não há mais sentido em continuar executando aquela determinada tarefa. Porém, mesmo que ocorra o não cumprimento de um Prazo não-Crítico, o sistema ainda pode continuar a executar aquele processo, já que ainda existe uma possibilidade remota de que o processo conclua sua execução em tempo de seus resultados serem utilizados.

Em alguns momentos é viável se conhecer o Tempo de Pior Caso de Execução de uma Tarefa (*Worst Case Execution Time* ou WCET). A medida deste tempo pode ser feita medindo manualmente os ciclos e loops ou estimando através de Ferramentas de Análises. Mas o cálculo deste parâmetro é incerto devido a variação de atraso decorrente de Memórias Cache, Interrupções, DMA, Latências no Barramento, Latências na Rede, entre outros.

Também podem ser obtidos o Período e a Freqüência em que uma tarefa é colocada em execução. O Período (T) é determinado pelo intervalo de tempo em que o processo estará habilitado a iniciar (ou continuar) sua execução no processador. A freqüência (F) pode ser obtida através da relação F=1/T.

2.6 Herança de Prioridades

Ocasionalmente, um processo de alta prioridade pode atingir o estado de bloqueado à espera de um processo de baixa prioridade. Tal efeito é denominado por [25] como Inversão de Prioridade.

Um dos meios de limitar o efeito de Inversão de Prioridades é através da utilização da técnica de Herança de Prioridades descrito por [26]. Com a herança de prioridade, a prioridade de um processo passa-se não mais a ser um valor estático. Se um processo p de (alta prioridade) está suspenso à espera de um processo q (de baixa prioridade) para retomar sua execução, então o processo q irá assumir a mesma prioridade do processo p.

Com isto, ao utilizar a técnica de Herança de Prioridades, a prioridade de um determinado processo é calculada através do valor máximo entre a prioridade do próprio processo e a prioridade de todos os demais processos que possuem dependência temporal deste processo.

2.7 Priority Ceiling

Segundo o protocolo *Immediate Ceiling Priority Protocol* [27,28,29,30], cada recurso possui um determinado valor de *ceiling*, o qual representa o valor máximo da prioridade entre todos os processos que o usam. E a cada processo são designados: uma Prioridade Estática, determinada em tempo de projeto, e uma Prioridade Dinâmica, determinada pelo máximo entre sua própria prioridade e o valor do *ceiling* dos recursos associados a ele.

Neste trabalho, o protocolo Immediate Ceiling Priority será empregado nas trocas de mensagens por uma rede de comunicação. Neste caso, os recursos utilizados serão as mensagens. Deste modo, ao estabelecer uma comunicação entre dois processos, a mensagem a ser transmitida receberá como ceiling o valor da maior prioridade dos processos interlocutores e o processo de menor prioridade receberá o valor do *ceiling* da mensagem como sua Prioridade Dinâmica.

2.8 Conclusões do Capítulo

Neste capítulo foram exibidos os princípios nos quais um sistema de tempo real se baseia, assim como suas classificações e aplicações. Porém, é necessário que exista um componente encarregado do gerenciamento de recursos do sistema, alocação das fatias de tempo de processamento para cada processo e da interação destes processos.

Este componente deve estar sendo executado juntamente com as demais tarefas e possuir um tempo de resposta rápido para todas as operações necessárias pelo sistema. O componente que está sendo descrito acima é denominado Sistema Operacional.

Capítulo 3
Sistemas Operacionais

Segundo [11], um sistema operacional consiste em um conjunto de rotinas com o objetivo de gerenciar os recursos presentes no sistema. Algumas dessas rotinas são as Chamadas de Sistema (ou *System Calls*) que possibilitam o acesso aos serviços de um sistema operacional de modo protegido e seguro.

Neste capítulo serão analisadas algumas das características, estruturas e primitivas de sistemas operacionais, direcionando o enfoque e as abordagens apresentadas em Sistemas Operacionais de Tempo-Real (RTOS).

3.1 Processo

Um processo é composto por um programa executável, assim como valores de contador de programa, variáveis do programa e registradores. Deste modo, cada processo é executado em sua própria CPU Virtual.

Tais processos podem ser executados em paralelo em estações com múltiplos processadores, em um *hardware* fortemente acoplado; ou em *clusters* de PCs, identificando um hardware MIMD fracamente acoplado. A definição das categorias de computadores de Múltiplas CPUs está localizada no Capítulo de Computação Distribuída. Eles também podem executar em pseudoparalelismo em um mesmo processador, através de mudanças de contexto entre um processo e outro.

3.1.1 Mudança de Contexto

A finalidade da mudança de contexto é chavear o processo em execução, através da troca do processo que está sendo executado por um outro. Uma rotina de interrupção é classificada por [12] como a intervenção do sistema operacional a um determinado evento que ocorreu. Antes que ocorra a mudança de contexto, o conteúdo dos registradores é salvo, e então o processo é tirado de execução.

Então, momentos antes do processo voltar à execução, tal conteúdo armazenado anteriormente é recuperado aos devidos registradores, como pode ser observado na Figura 1.

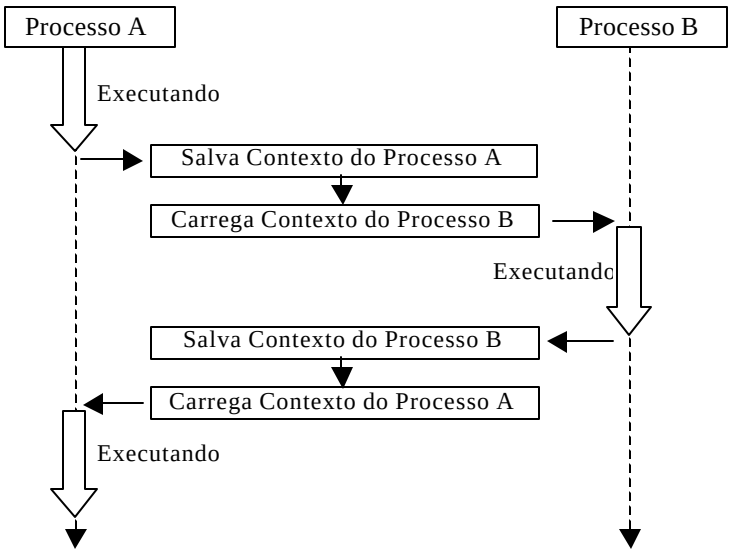

Figura 1: Mudança de Contexto

Como o instante que ocorre a mudança de contexto é determinado pelo Sistema Operacional, e não pelo Processo, uma eventual reprodução da execução de tais processos pode não obter um resultado idêntico. Isto é, uma mudança de contexto pode ocorrer em instantes diferentes para cada vez que os processos são executados.

3.1.2 Estados dos Processos

Em sistemas *time-sharing* um processo não é executado sem interrupções. Sendo assim, ele pode sair de execução e ceder o tempo de processamento a outro processo; ele pode ser bloqueado à espera de um recurso; ou simplesmente ele pode terminar.

Deste modo, é possível identificar 3 estados básicos dos processos: Executando (*Running*), Bloqueado (*Blocked*), e Pronto (*Ready*). Entre estes estados podem

ocorrer 4 transições básicas: (1) o processo pode ser bloqueado à espera de um recurso, (2) o escalonador recoloca o processo em condição de Pronto, (3) o escalonador coloca um processo pronto em execução, (4) o recurso esperado torna-se disponível. Esta representação pode ser observada na Figura 2.

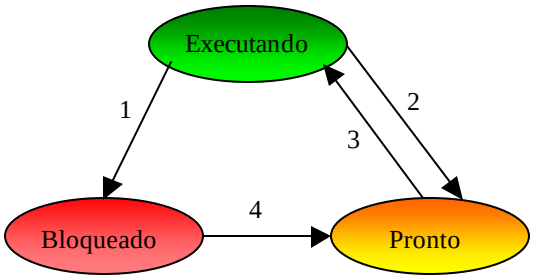

Figura 2: Representação Gráfica dos estados e transições dos processos

3.2 Comunicação Inter-Processos

Freqüentemente, processos que trabalham para a solução de um mesmo problema necessitam compartilhar algum recurso e, dependendo da lógica do algoritmo implementado nos processos, pode ocorrer um problema nomeado como *Race Conditions* [11].

Race Conditions são causado pela condição de disputa para a utilização de um mesmo recurso por dois processos concorrentes. Este problema pode gerar inconsistência de dados entre os processos, caso ambos utilizem o mesmo recurso simultaneamente. As próximas seções contêm alguns modos de manipular este problema.

3.2.1 Região Crítica

Com o intuito de se evitar problemas com recursos compartilhados é necessário garantir a Exclusão Mútua de processos que utilizam um recurso comum. Deste modo, a exclusão mútua é descrita por [12] como um modo de impedir que dois ou mais processos acessem um mesmo recurso no mesmo instante. Ou seja, se um processo está utilizando um determinado recurso compartilhado e um outro

processo necessita deste recurso, o segundo processo será colocado em espera até a liberação do recurso.

3.2.2 Desativação de Interrupções

Um dos modos mais simples de se fazer uma Exclusão Mútua é impedindo que ocorra a mudança de contexto. Para isto, as interrupções podem ser desativadas [11,12] antes que o processo entre na região crítica e são re-ativadas depois que o processo sair da região crítica.

A desvantagem da utilização deste mecanismo está na segurança, pois um processo que desativou as interrupções pode não voltar a ativá-las. Esta situação impedirá que o sistema volte ao seu funcionamento normal.

Por outro lado, o Sistema Operacional pode fazer uso deste mecanismo para a manipulação de estruturas internas como o acesso à lista de processos, garantindo, assim, a consistência de seus dados.

3.2.3 Sleep Wake-up

A primitiva *Sleep* [11] mantém um processo bloqueado à espera de um *WakeUp* vindo de um outro processo, ou de algum módulo do Sistema Operacional. Esta Chamada de Sistema também pode ser implementada de modo a manter o processo bloqueado por um certo intervalo de tempo, determinado na chamada de *Sleep*. Depois de passado este intervalo de tempo, o Sistema Operacional gera um Sinal de *Wakeup*, o qual volta o processo àlista de Prontos.

3.2.4 Semáforos

A sugestão de se utilizar um semáforo [17] surgiu a partir da utilização de uma variável inteira como um contador para controlar o bloqueio do semáforo. Sendo assim, as chamadas de sistema *Sleep* e *WakeUp* foram generalizadas em

procedimentos *Up* e *Down*, os quais gerenciariam o incremento e decremento desta variável.

A primitiva *Down* checa se o valor inteiro é maior que zero. Neste caso, a variável é decrementada e é possível entrar na Região Crítica. Caso contrário (variável com valor zero), o processo é bloqueado.

Já a primitiva *Up* indica que o processo saiu da Região Crítica. Em sua chamada, a variável é incrementada. Caso exista algum processo bloqueado pelo semáforo ele é acordado pelo sistema e colocado em execução.

3.2.5 Passagem de Mensagem

A passagem de mensagem pode ser utilizada para fazer comunicação e sincronização de processos. Suas primitivas são nomeadas por muitos autores como *Send* e *Receive* [11,12], e sua comunicação pode ser síncrona ou assíncrona, direta ou indireta, como demonstrado na Figura 3.

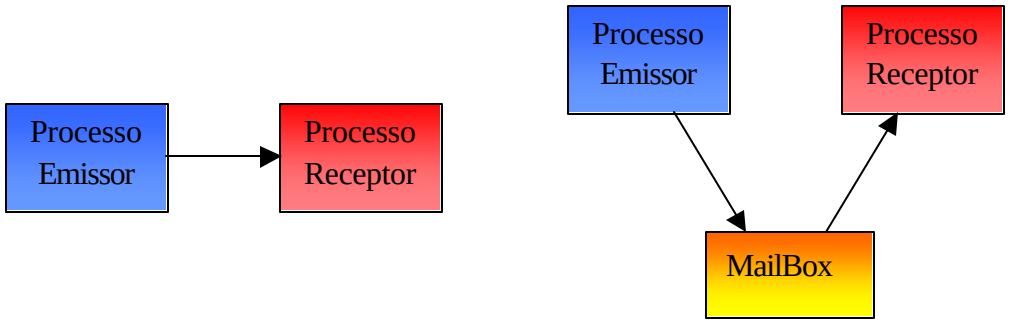

Figura 3: Trocas de Mensagem Direta e Indireta

Em uma comunicação direta, a passagem de mensagem é feita apenas do emissor para o receptor, sem a necessidade de utilização de um *buffer* intermediário. Uma desvantagem desta utilização é a especificação dos números (ou nomes) dos processos envolvidos.

Já com a utilização de comunicação indireta, um *buffer* compartilhado (também conhecido como *MailBox*) é utilizado, onde o processo emissor pode depositar suas mensagens e o processo receptor pode retirá-las. Com este tipo de troca de mensagens não há a necessidade de se especificar a identificação dos processos envolvidos na comunicação.

Em uma comunicação síncrona a primitiva *Send* fica bloqueada enquanto o buffer de envio estiver cheio e a primitiva *Receive* fica bloqueada enquanto o buffer de recebimento estiver vazio. A vantagem deste tipo de implementação é a possibilidade de estabelecimento de um ponto de sincronismo entre os dois processos. Sua principal desvantagem é a não ocorrência do paralelismo durante o tempo que um dos processos fica bloqueado àespera de outro.

Por outro lado, em uma comunicação assíncrona, nem o emissor e nem o receptor ficam bloqueados com as primitivas. Sendo assim, existe a necessidade de utilização de um *buffer* ou um *MailBox* para o armazenamento das mensagens, assim como mecanismos para garantir o envio e o recebimento das mensagens.

3.2.6 Sinais

Um sinal é uma mensagem muito pequena que pode ser enviada a um processo ou a um grupo de processos. A única informação dada para um processo é o número de identificação do sinal: não há espaço nos sinais padrão para argumentos, ou uma mensagem, ou outra informação complementar.

Sinais servem dois propósitos principais: alertar um processo que um evento específico ocorreu; e forçar um processo a executar uma função tratadora de sinal incluída em seu código. Claro que os dois propósitos não são mutuamente exclusivos, uma vez que geralmente um processo deve reagir a algum evento executando uma determinada rotina.

3.2.6.1 Sinais em tempo real

O padrão POSIX [24] introduziu uma diferente classe de sinais chamado sinais de tempo real. A diferença entre sinais padrões e sinais em tempo real é que múltiplos sinais em tempo real podem ser enviados, ou seja, os sinais de tempo real do mesmo tipo podem ser tratados como fila. Isso assegura que múltiplos sinais enviados serão recebidos.

Programas executados em modo usuário podem receber e enviar sinais. Isso significa que um grupo de operações deve ser definido para que isso possa acontecer. Infelizmente, devido a razões históricas, varias chamadas de sistemas incompatíveis existem para servir exatamente ao mesmo propósito.

3.3 Escalonadores

Os escalonadores de processos permitem ao sistema operacional decidir qual o próximo processo a entrar em execução. Os objetivos da utilização dos escalonadores são: garantir que cada processo ganhe um tempo de processamento, manter a CPU ocupada executando processos, diminuir o tempo de resposta (*turnaround*) e aumentar a taxa de execução de processos (*throughput*).

Os tipos de escalonamento podem ser classificados em preemptivos e não preemptivos. Os escalonamentos não preemptivos colocam um processo em execução e o processador somente será liberado para outro processo a partir do término do atual.

Já os escalonadores preemptivos permitem que o sistema operacional interrompa um processo em execução e coloque um outro para ser executado. Esta troca de processos pode ocorrer devido ao surgimento de um processo com maior prioridade, como em um sistema de tempo-real, ou simplesmente devido ao

término de uma fatia de tempo reservado a um determinado processo. Esta troca de processamento é feita através da mudança de contexto, vista anteriormente.

3.3.1 Escalonamento Round Robin

O escalonamento Round Robin é utilizado principalmente em sistemas *time-sharing*. Ele consiste de uma fila de processos, onde cada processo é executado durante um tempo fixo (*quantum* ou *slice*). Após expirar este tempo, ocorre uma preempção, e o processo corrente passa a ocupar a última posição na fila de processos e o primeiro processo da fila assume a execução. Um exemplo deste algoritmo pode ser observado na Figura 4.

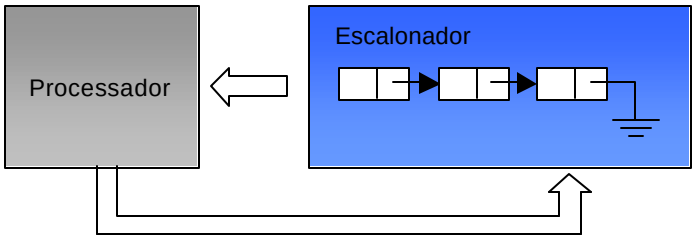

Figura 4: Escalonamento Round Robin

3.3.2 Escalonamento Baseado em Prioridade

Não são em todos os casos que os processos podem ser tratados de forma igualitária como acontece no algoritmo Round Robin. Sendo assim, no escalonamento baseado em prioridade, é atribuída uma prioridade para cada processo e o próximo processo a ser escolhido pelo escalonador será o de maior prioridade. No caso em que um processo com maior prioridade do que o em execução seja criado no sistema, ocorrerá uma preempção que removerá o processo corrente e colocará o processo de maior prioridade em execução.

Um problema freqüente neste algoritmo é que processos com altas prioridades podem ocupar todo o tempo de processamento e impedir processos com menores prioridades de executar. Sendo assim, um modo de solucionar este problema é

atribuir a prioridade dinamicamente a processos, implementado este algoritmo de modo a decrementar as prioridades dos processos em execução.

3.3.2.1 Escalonamento Baseado em Taxa Monotônica de Prioridades

O Escalonamento baseado em Taxa Monotônica de Prioridades é responsável por designar prioridades para processos periódicos. A relação entre período de um processo e prioridade pode ser determinada abaixo:

$$Ti > Tj \rightarrow Pi < Pj$$

Isto significa que processos que são executados em períodos (T) mais curtos (ou seja, com maior freqüência) serão escalonados com maior prioridade(P). Já para aqueles cuja execução ocorre com menor periodicidade, uma menor prioridade será determinada.

Para processos cujo Deadline coincida com o Período, o cumprimento do deadline de cada processo é garantido por [20] caso a média das taxas de computação por período de todos processos for menor que a raiz N-ésima de 2, menos 1, onde N é o número de processos no sistema.

$$\sum_{i=1}^{N} C_i/T_i < N\,(2^{1/N} - 1)$$

Através da expressão acima, é possível garantir o cumprimento das deadlines do sistema sempre que para uma quantidade N de processos, a taxa de utilização for menor que U, conforme a tabela abaixo:

Número de Processos (N)	Utilização do Sistema (U%)
1	100
2	82.8
3	78.0
4	75.7
5	74.3
10	71.8

3.3.3 Escalonamento Baseando em Deadline

Uma outra possibilidade de escalonamento é a utilização de um Prazo associado a cada processo. Tal Prazo é determinado através de um valor numérico referente ao máximo instante que um processo pode finalizar sua execução sem causar o não cumprimento de um requisito temporal.

Deste modo, o Escalonamento por Prioridade concede o tempo de execução aos processos cujo prazo para sua finalização estão mais próximos.

3.3.3.1 Escalonamento Baseado em Taxa Monotônica de Deadlines

Como visto anteriormente, o escalonamento baseado em Taxa Monotônica de Prioridades atinge seu ótimo quando o Prazo (ou *Deadline*) de um processo coincide com o seu Período. Para os demais casos, foi definido por [19] o Escalonamento Baseado em Taxa Monotônica de *Deadlines*.

Portanto, para quando o Prazo não coincidir com o Período, Leung e Whitehead definem que o Escalonamento baseado em Taxa Monotônica de *Deadlines* é responsável por designar Prioridades baseadas nos Prazos dos processos. A relação entre o prazo (D) de um processo e sua prioridade (P) pode ser determinada abaixo:

$$Di > Dj \rightarrow Pi < Pj$$

Isto significa que processos com prazos mais próximos serão escalonados com maior prioridade. Já para aqueles cujo prazo está mais distante, uma menor prioridade será determinada.

3.3.4 Escalonamento Baseado em Menor Latência

Este algoritmo de escalonamento tem o objetivo de escolher o processo de menor latência para ser o próximo processo a ser executado. A latência, neste caso, é o tempo de execução restante ao pior caso de execução de um processo que termine em tempo de cumprir seu Prazo.

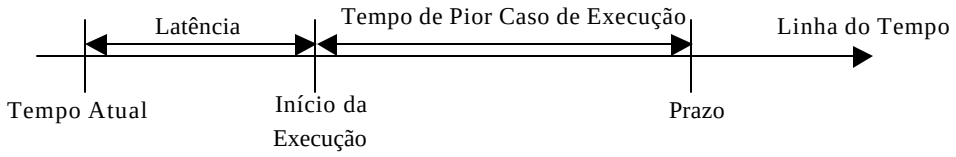

Figura 5: Exemplificação de Execução

Como pode ser observado na Figura 5, o Início de Execução é o valor numérico correspondente ao tempo em que é dado o início da execução do processo. Já o cálculo da Latência é realizado através da subtração do Prazo (*Deadline*) pela soma do Tempo de Pior Caso de Execução (*Worst Case Execution Time*) e do Tempo Atual.

Uma das principais dificuldades deste algoritmo é saber o tempo que dura a execução de um determinado processo em seu pior caso. O tempo pode ser calculado a partir de estimativas por uma simulação, ou através de uma abordagem analítica onde se pode construir um modelo matemático para o processo.

3.3.5 Conclusões sobre Escalonamento

Podemos observar nesta seção que apesar do algoritmo de escalonamento Round Robin não possuir nenhum requisito temporal, ele é eficiente para sistemas *time sharing*. Este escalonamento também exemplifica os objetivos de um escalonador com muita simplicidade.

Já o escalonamento por prioridade pode ser empregado em sistemas de tempo-real não críticos. Outros requisitos temporais podem ser mapeados sobre a prioridade, como Prazo e a Latência.

Por outro lado, algoritmos baseados em prazo ou latência são indicados para sistemas críticos, apesar de algumas dificuldades como calcular o tempo de pior caso de execução de um processo.

3.4 Deadlock

Um *Deadlock* pode ser caracterizado como um estado de impasse. Ele é formalmente definido por [11] como “um conjunto de processos está em estado de *Deadlock* se cada um dos processos está esperando um evento que somente um outro processo deste conjunto pode gerar”.

Foi demonstrado por [18] que quatro condições devem existir para haver um estado de impasse:
1- Existência de recursos, sendo que cada qual pode estar designado a somente um processo ou disponível.
2- Processos que atualmente estão alocando um recurso já pediram pela alocação de um outro recurso previamente.
3- Um recurso não pode ser desalocado de um processo. Deve-se esperar que o processo o libere.
4- Deve haver uma cadeia de dois ou mais processos, cada qual esperando a liberação de um recurso alocado por um outro processo da cadeia.

3.5 Conclusões do Capítulo

Neste capítulo foram analisadas algumas características de um sistema operacional. Entre elas foi possível observar alguns algoritmos utilizados para o escalonamento de processos e métodos para a comunicação interprocessos.

Nos próximos capítulos haverá um foco maior na computação distribuída. Porém, antes de continuar discutindo sobre distribuição, é necessário que sejam analisados alguns protocolos de uma rede de comunicação, assim como alguns empecilhos para que esta comunicação ocorra. Tais assuntos serão levantados no capítulo a seguir.

Capítulo 4
Rede de Comunicação

De modo a permitir que o desenvolvedor de uma Aplicação de Tempo Real utilize a computação distribuída, é necessário que um sistema forneça suporte à comunicação de rede.

Para isto, com base em [14], [15] e [16], serão analisados neste capítulo os principais protocolos de rede utilizados atualmente, com o objetivo de prover a comunicação em rede através de chamadas de sistema do sistema operacional.

4.1 Protocol Data Unit

Para reduzir a complexidade de projeto, a maioria das redes está organizada como uma série de camadas. Cada uma é responsável por certo grau de detalhes na comunicação, onde se oferecem serviços para as camadas superiores. Através de abstrações e independência de implementação, não há a necessidade da camada conhecer a implementação da camada inferior.

As iterações entre camadas adjacentes são realizadas na forma de solicitação e resposta. Essas iterações são oferecidas nos Pontos de Acesso ao Serviço (SAP) [14] e podem ser dos tipos: *request*, *indication*, *response* e *confirmation*.

Entidades pares são aquelas que estão dentro de uma mesma camada. As informações trocadas entre essas unidades são chamadas Unidades de Dados do Protocolo (PDU) [14]. As PDUs trocadas entre entidades adjacentes são transmitidas de forma encapsulada.

4.2 Camada de Rede - IP- Internet Protocol

Conhecido por suas iniciais IP, *Internet Protocol* [15] é um mecanismo de entrega sem conexão e não confiável. Três importantes definições são providas de IP:

- O software de IP executa funções de roteamento escolhendo o caminho pelo qual os dados serão enviados.
- Utiliza os serviços providos pela camada de Enlace e provê à Camada de Transporte entrega de pacotes de forma não confiável.

Entre as regras descritas no último item, podemos citar, como *hosts* e roteadores devem processar os pacotes, como e quando mensagens de erro devem ser geradas e em quais condições os pacotes devem ser descartados.

Um datagrama IP [22] consiste de uma área de cabeçalho, ilustrado na Figura 6, e uma área de dados. Este cabeçalho tem uma parte fixa de 20 bytes e uma parte opcional de tamanho variado.

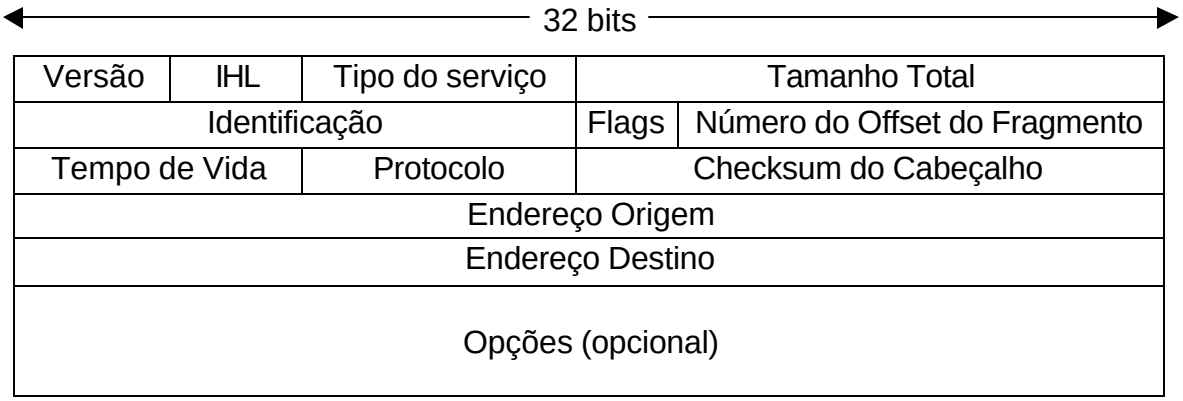

Figura 6: Cabeçalho IP

O campo Versão indica a versão de protocolo.

Como o tamanho do cabeçalho não é constante, o campo IHL indica o tamanho do cabeçalho.

O campo Tipo de Serviço permite ao *host* dizer à sub rede que tipo de serviço ele quer, como transmissão rápida e sem garantia.

O campo de Tamanho Total indica o tamanho de todo o datagrama, isto é, cabeçalho + dados; seu tamanho máximo é de 64 bytes.

O campo Identificação é utilizado para determinar a qual datagrama pertence um fragmento. O valor utilizado em Identificação deve ser único para cada par origem-destino. Ligado a este campo, está o campo Número do *Offset* do Fragmento, utilizado para a remontagem dos datagramas fragmentados.

Entre os *Flags* podemos citar*: Don't Fragment*, que impede que roteadores intermediários fragmentem o pacote com este bit habilitado, e *More Fragments*, sendo que todos os fragmentos contêm este bit habilitado, exceto o último.

Tempo de Vida é um contador para limitar o tempo de vida de pacotes. A unidade de tempo de vida é definida em segundos por [32], e é decrementado a cada elemento de rede pelo qual o pacote passa. Quando este campo atinge o valor zero o pacote é descartado.

O campo Protocolo indica o protocolo da informação que está contido neste pacote.

O campo *Checksum* do Cabeçalho visa validar a integridade do Cabeçalho IP. O algoritmo de *checksum* é simples: somar todo o cabeçalho de 16 bits em 16 bits utilizando complemento um.

Os campos Endereço Origem e Endereço destino indicam o endereço de rede da estação que enviou este pacote e da estação que irá recebê-lo.

O campo Opções tem seu uso opcional. Nele podem existir uma ou mais opções, que são classificadas em:

- Caso 1: Contém um simples octeto com o tipo da opção;
- Caso 2: Contém um octeto com o tipo da opção, um octeto com o tamanho e os demais octetos com os dados desta opção.

4.3 Camada de Transporte

Nesta seção veremos os dois mais importantes protocolos da camada de transporte utilizados atualmente: TCP e UDP. Também será tratada a sobrecarga da comunicação devido ao estabelecimento de conexões. Ainda nesta seção serão vistas as implementações destes dois protocolos sobre *socket*.

4.3.1 Sockets

Socket é um mecanismo básico para comunicação entre aplicações. Um *socket* [15] é um valor numérico utilizado para identificar o ponto final de uma comunicação de um determinado processo em uma determinada máquina sobre uma rede de computadores. Devido ao fato da utilização de *sockets* ser bastante simples e flexível, isto facilita o desenvolvimento de aplicações distribuídas.

4.3.2 TCP - Transmission Control Protocol

O protocolo TCP [22], ou *Transmission Control Protocol*, foi projetado para prover uma comunicação de fluxo de *bytes* fim-a-fim confiável sobre uma rede não confiável. Dentre suas características:

- TCP especifica um formato de dados e reconhecimentos que dois computadores trocam para obterem uma transferência confiável.
- TCP especifica procedimentos para garantir que os dados cheguem corretamente.
- TCP distingue múltiplos destinos em uma mesma máquina.
- TCP provê recuperação de erros entre máquinas interlocutoras.
- TCP especifica como duas máquinas iniciam uma transferência em forma de fluxo e como ambas as partes concordam que esta terminou.

O serviço TCP é obtido tendo ambos, emissor e receptor, criado pontos finais (*endpoints*) chamados *Sockets*. Cada Socket tem um número de identificação e armazena o IP do *host* e um inteiro de 16 bits chamado de número da porta. A

porta juntamente com um endereço IP forma um inteiro de 48 bits único chamado de TSAP.

Todas conexões TCP são full-duplex, isto é, o tráfego pode ir a ambas as direções ao mesmo tempo, e ponto-a-ponto, isto é, cada conexão tem exatamente 2 pontos finais. Deste modo, TCP não suporta Broadcasting e nem Multicasting.

Assim como no datagrama IP, o segmento TCP [14] consiste de uma área de cabeçalho, ilustrado na Figura 7, e uma área de dados. Este cabeçalho tem uma parte fixa de 20 *bytes* e uma parte opcional de tamanho variado.

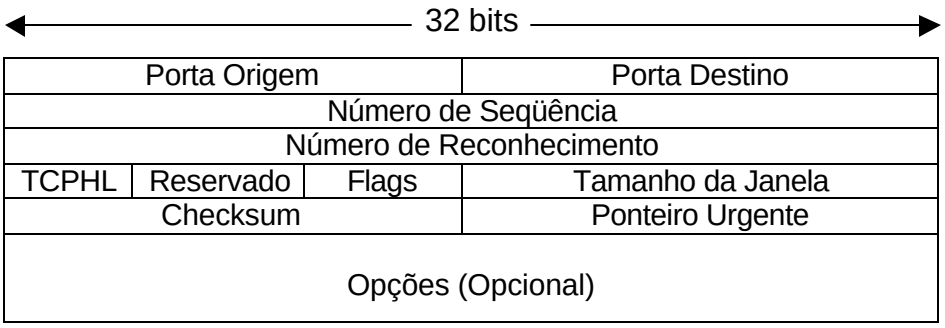

Figura 7: O cabeçalho TCP

Os campos Porta Origem e Porta Destino identificam os Pontos finais da conexão. Cada *host* pode decidir por ele mesmo como alocar suas próprias portas a partir de 1024. Isto por que as portas até 1024 são ditas "conhecidas" e reservadas para alguns protocolos e aplicações, como Telnet, FTP, HTTP, entre outros.

Os campos Número de Seqüência e Número de Reconhecimento existem devido a toda PDU ser numeradas em TCP. Tal numeração será vista posteriormente neste capítulo.

Já TCPHL (*TCP Header Length*), um campo de 4 bits, indica quantas palavras de 32 bits existem no cabeçalho TCP. Isto é necessário devido ao campo Opções ter tamanho variável.

O campo Reservado não é utilizado. Ele foi reservado para futuras modificações no protocolo.

O Campo Flags é composto por:

- URG: Habilitado quando o campo Ponteiro Urgente estive em uso.
- ACK: Habilitado para indicar que o valor em Número de Reconhecimento é válido.
- PSH: Indica que este segmento requer entrega de dados (*Push*).
- RST: É utilizado para reiniciar a conexão devido a alguma falha.
- SYN: É utilizado para estabelecer conexões.
- FIN: É utilizado para liberar uma conexão.

O Controle de Fluxo em TCP é manipulado através do algoritmo de Janela Deslizante [23]. Assim, o campo Tamanho da Janela indica quantos *bytes* podem ser enviados através do canal reverso.

O campo *Checksum* visa validar a integridade do cabeçalho TCP.

4.3.2.1 Estabelecimento de Conexões

Para que haja uma concordância com o número da seqüência a ser utilizado, as conexões são estabelecidas em TCP através de *Three-Way Handshake* [14] como pode ser observado na Figura 8. Este protocolo de estabelecimento de conexão não requer que ambos os interlocutores comecem enviando o mesmo número de seqüência.

O procedimento normal começa com a estação cliente enviando um pedido de conexão SYN à estação servidora, contendo um número de seqüência X. A estação servidora, por sua vez, responde SYN+ACK com um número de seqüência Y e um número de reconhecimento X+1. Finalmente, a estação cliente reconhece a escolha da estação servidora enviando a primeira ACK com número de seqüência X+1 e número de reconhecimento Y+1.

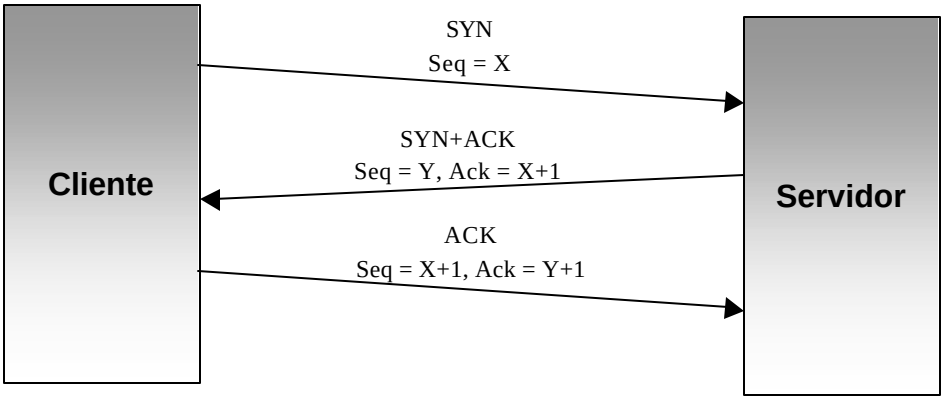

Figura 8: Protocolo Three-Way Handshake

4.3.2.2 Utilização de TCP em Socket

A implementação de um cliente e um servidor utilizando *sockets*, como apresentado pela Figura 9, dá-se através das primitivas: socket(), o qual cria um novo ponto de comunicação; bind(), que associa um endereço a uma determinada porta; listen(), função pela qual é informada a disponibilidade do servidor de aceitar novas conexões; accept(), mantém o servidor bloqueado até o recebimento de um pedido de conexão; connect(), função onde o cliente tenta o estabelecimento de conexão com um determinado servidor; send(), envia um fluxo de bytes através de uma conexão; receive(), recebe um fluxo de dados enviado pela função send(); e close(), libera a conexão.

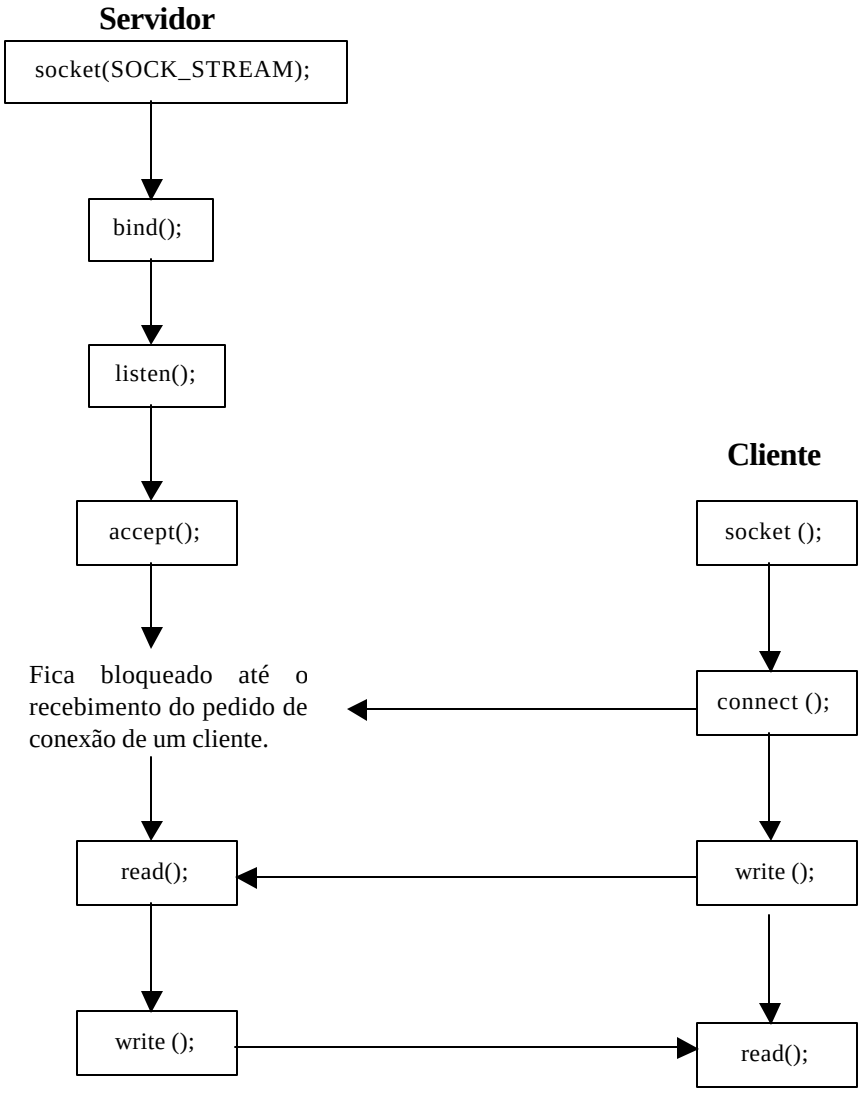

Figura 9: Modelo de Comunicação Orientada a Conexão

4.3.3 UDP - User Datagram Protocol

O protocolo da camada de rede IP também suporta sobre ele um protocolo da camada de transporte sem conexão: UDP [15], *User Datagram Protocol*. UDP dispõe um modo de aplicações enviarem PDUT sem ter estabelecido uma

conexão. Um segmento UDP [14] consiste de um cabeçalho de 8 *bytes* seguido de Dados assim como demonstra a Figura 10.

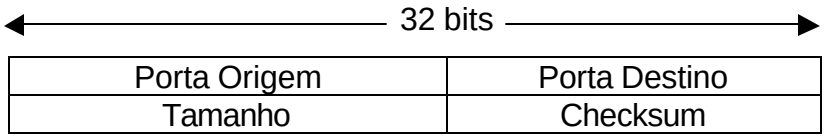

Figura 10: O cabeçalho UDP

Assim como TCP, os campos Porta Origem e Porta Destino identificam os Pontos finais locais da conexão. O campo Tamanho inclui o cabeçalho de 8 *bytes* e os dados. O campo *Checksum* baseia-se na soma de um pseudocabeçalho e dos dados de 16 bits em 16 bits utilizando complemento um. Ele é opcional e se o valor armazenado for zero ele não é computado.

4.3.3.1 Utilização de UDP em Socket

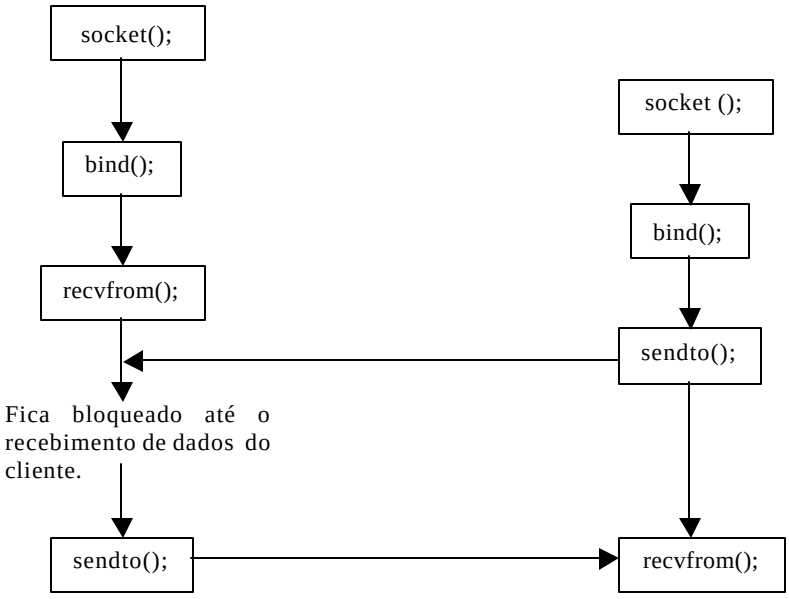

Figura 11: Modelo de Comunicação Não Orientada a Conexão

A implementação de um cliente e um servidor UDP utilizando *sockets*, como apresentado pela Figura 11, dá-se através das chamadas: socket(), a qual cria um novo ponto de comunicação; bind(), que associa um endereço a uma determinada

porta; sendto(), envia um conjunto de bytes dentro de uma PDUT; recvfrom(), recebe dados enviados pela função sendto().

4.4 Incertezas Quanto ao Tempo de Transmissão

Alguns aspectos que possivelmente gerem atraso durante uma transmissão podem ser levantados. Os protocolos de comunicação atualmente utilizados foram desenvolvidos em múltiplas camadas, para permitir maior flexibilidade. Em cada camada existem *buffers* e identificadores de cabeçalho, pacotes são fragmentados e unidos novamente. Esses aspectos podem causar um atraso variável e difícil de ser especificado.

Na camada física devemos ressaltar a Largura de Banda, a qual pode se tornar um gargalo na transmissão de um grande fluxo de dados. Também, a velocidade de transmissão entre os interlocutores é limitada em 300.000km/s, pois nenhum sinal elétrico ou luminoso ultrapassa velocidade da luz. A Topologia é um outro fator importante, já que a cada estação intermediária deve-se considerar o tratamento e alteração dos cabeçalhos, roteamento e filas para a transmissão.

Já na camada de enlace devemos nos preocupar com possíveis Colisões e Retransmissões e a espera pela utilização de um Meio Ocupado. Também é possível que ocorra um atraso durante a Leitura e Escrita em *Buffers*, Codificação e Decodificação de Cabeçalhos e a Troca de PDUs entre camadas.

Por sua vez, na camada de rede, pode ocorrer atraso durante a Leitura na Tabela de Rotas, Codificação e Decodificação de Cabeçalhos, Fragmentação e Junção de Pacotes e a Troca de PDUs entre camadas.

E na implementação de alguns protocolos de Transporte como TCP ainda existe o tráfego de Mensagens de Controle, como por exemplo, *ACKs*, *Three-Way-Handshake*, Finalização de Conexão, entre outros.

4.5 Conclusões do Capítulo

Devido a UDP não necessitar estabelecer conexão e nem liberar conexão, se o custo de uma mensagem UDP durar 1 unidade de tempo, então o custo de uma mensagem TCP irá durar 7 unidades, sendo 3 durante o estabelecimento da comunicação (Three Way Handshake), 2 para o envio de dados e 2 para liberar a conexão.

Por outro lado, uma das vantagens de TCP sobre UDP é que UDP não possui garantia de entrega. Também, ao contrário de UDP, TCP possui mecanismos de tratamento e recuperação de erros.

Capítulo 5
Computação Distribuída

A computação distribuída pode ser vista como uma decomposição da aplicação principal para que esta seja distribuída numa rede de computadores ou processadores realizando tarefas cooperativas. Entre as vantagens para a decomposição das tarefas em módulos menores de forma a executarem distribuidamente podemos citar:

- Execução em paralelo, possibilitando resolver grandes problemas sem a necessidade de grandes computadores;
- Alocação, gerenciamento de grandes conjuntos de tarefas em um conjunto de estações de trabalho.
- Balanceamento de carga de trabalho entre as estações, através de migração de processos.
- Prover transparência ao usuário em relação à distribuição de tarefas e dados.
- Múltiplos computadores interligados em rede podem ser usados pelos sistemas para tratamento de tolerância a falhas.
- Compartilhamento de Recursos entre as estações (como impressoras, discos).

5.1 Classificação de Computadores com Múltiplas CPUs

Atualmente, a classificação para computadores de Múltiplas CPUs mais utilizada foi proposta por [21], onde as classes de multi computadores são classificadas em número de instruções e número de dados: SISD (*Single Instruction Single Data*), SIMD (*Single Instruction Multiple Data*), MISD (*Multiple Instruction Single Data*) e MIMD (*Multiple Instruction Multiple Data*).

5.2 Características dos Modelos Integradores

Estes modelos tentam resolver dois tipos de problemas:

1. Identificar a compatibilidade entre tipos definidos em modelos diferentes e fazer o mapeamento entre eles;
2. Compatibilizar o modelo de execução de objetos implementados em arquiteturas diferentes e que precisam interagir.

No projeto de um destes modelos integradores devem ser considerados:

1. O conjunto de modelos a serem integrados, incluindo linguagens de diferentes paradigmas de programação;
2. A maneira como os modelos são integrados, adotando uma abordagem de união ou de interseção das características das linguagens integrantes;
3. Grau de extensibilidade do modelo integrador; ele deve prover mecanismos que lhe permitam incorporar novas características;
4. Grau de flexibilidade e transparência do ponto de vista dos modelos integradores, tanto para o desenvolvedor de objetos quanto para o seu usuário cliente.

5.3 CORBA

A idéia por trás da arquitetura CORBA é a de se ter um software intermediário que manipule e faça os acessos de requisições sobre conjuntos de dados. Este intermediário é referido como um *Object Request Broker* (ORB), como mostra a Figura 12. O ORB interage e faz requisições para objetos diferentes. Ele permanece na máquina servidora um nível abaixo da camada de aplicação. Um ORB negocia entre mensagens de requisições de objetos ou servidores de objetos e conjuntos de dados associados.

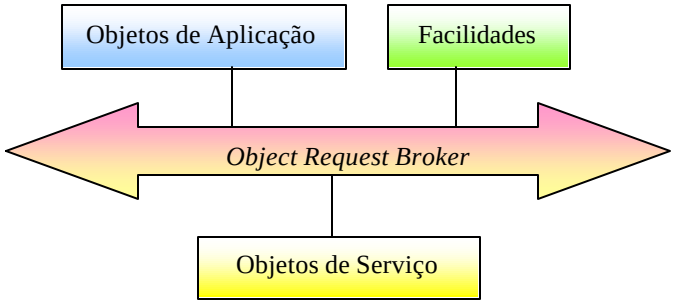

Figura 12: A Arquitetura CORBA

O paradigma da arquitetura CORBA segue uma combinação de duas metodologias existentes. A primeira é a computação distribuída cliente/servidor. Ela é baseada em parte nos sistemas por passagem de mensagens (*message-passing*), encontrada principalmente no ambiente UNIX. A segunda metodologia é a programação orientada a objetos.

Os objetos CORBA, que são os processos que interligam os conjuntos de dados com outros objetos CORBA ou com clientes, provêem ligações através da interação com o ORB através de um *Object Adapter* especificado pelo OMG.

Um *Object Adapter* é um objeto CORBA que permite o acesso aos demais objetos. Ele permite que um ORB faça chamadas de métodos de objetos e também que crie e apague aquele objeto, bem como provê segurança para o ORB e mapeamento de referências de objetos para implementação. Ele permite também, que um ORB gere e utilize referências para esses objetos. O propósito é prover um outro nível de abstração, pois desta forma os ORBs podem interagir de um modo padrão com outros objetos CORBA em diferentes plataformas e redes.

Aplicações clientes podem contatar objetos CORBA para requisição de dados. Esses clientes utilizam os conjuntos de dados de diferentes objetos, mas não possuem seus próprios dados. Estes programas são conhecidos como *data-dependents*, pois operam baseados em outros conjuntos de dados. O ORB é o

mecanismo que manipula as interações entre as aplicações clientes e os objetos CORBA.

O ORB manipula as interações entre esses objetos se comportando como uma API (*Application Programming Inteface*) para uma chamada remota de procedimento (RPC – *Remote Procedure Call*) orientada a objetos. Em outras palavras, as regras de mapeamento padronizadas pelo OMG são seguidas para se construir esta API. A arquitetura CORBA foi projetada para ser mais robusta e simples do que as chamadas de RPC e bibliotecas existentes. Os conjuntos de dados são agora separados e independentes do desenvolvedor de aplicações. O programador faz requisições de códigos nos dados através do objeto CORBA, e o ORB manipula a passagem de mensagens e acessa os dados do objeto através das definições daquele objeto.

A presença da metodologia orientada a objetos em CORBA é resultado da necessidade, não da escolha. Em CORBA, são utilizadas três características básicas da programação orientada a objetos. A primeira é o polimorfismo entre objetos. O ORB produz objetos diferentes independentes e reutilizáveis para diferentes aplicações. A segunda característica é o encapsulamento de dados. Cada aplicação cliente não conhece nada sobre os dados que acessa; ela simplesmente faz requisições para o respectivo objeto através do ORB, e o objeto retorna os dados para a aplicação. Como terceira característica tem-se a herança de dados. Se uma descrição de um objeto é projetada para a comunicação com um ORB, qualquer objeto derivado daquele objeto pai irá preservar a interface de comunicação daquele pai.

A transparência dos servidores (ou *hosts*) também é mantida dentro da CORBA. Os ORBs podem acessar e fazer chamadas para diferentes objetos CORBA em várias máquinas. Se uma aplicação cliente está executando em uma máquina, o ORB daquela máquina está apto a localizar dados em diferentes lugares daquela

máquina ou em máquinas diferentes. Esta transparência é alcançada, na maior parte, através dos repositórios especificados pelo OMG.

Diferentemente de RMI (Sun), presente em Java, CORBA não esta ligada somente a uma única linguagem de programação, ou seja, pode se desenvolver aplicações distribuídas em CORBA utilizando as mais diferentes linguagens como, por exemplo, C, C++, Ada e Cobol.

Os serviços disponibilizados por CORBA são descritos por uma interface, escrita em uma linguagem chamada de *Interface Definition Language* (IDL)**.** IDL mapeia as linguagens mais populares e inclusive Java. CORBA permite que objetos façam requisições de objetos remotos, através da invocação de métodos e permite que dados sejam passados entre os objetos. Entretanto, CORBA permite apenas que tipos de dados primitivos sejam passados entre objetos.

5.3.1 CORBA RT

CORBA RT, definido como extensão da Especificação Formal de CORBA 2.3 (98-12-01) e de *Messaging Specifications* (98-05-05), é um conjunto de extensões de CORBA para equipar ORBs para serem utilizados como componentes de um Sistema de Tempo-Real.

Porem, CORBA RT não suporta todas as características providas por CORBA, já que de forma a garantir a previsibilidade no sistema, CORBA RT deixa de prover algumas funcionalidades não determinísticas de CORBA para suportar o desenvolvimento de Sistemas de Tempo Real.

O comportamento determinístico dos componentes de um sistema de Tempo-Real promove a previsibilidade de todo sistema. E de modo a garantir o cumprimento dos requisitos de Tempo-Real, o sistema deve ser previsível.

Desta maneira, uma das maiores limitações de CORBA RT é que ele foi projetado para trabalhar em um sistema fechado, não permitindo que outros elementos que não foram pré registrados utilizem o sistema.

Também são utilizadas prioridades fixas que são atribuídas a um conjunto estático de clientes e servidores. Sendo assim, o uso de prioridades fixas impossibilita a flexibilidade de um desenvolvedor de STRs durante a implementação de uma Aplicação de Tempo Real.

5.4 RMI

RMI (Sun) é um modelo para o desenvolvimento de aplicações distribuídas que se baseia na distribuição remota de objetos e no acesso a determinados métodos destes objetos por outros que podem estar localizados em diferentes máquinas.

Este modelo foi projetado para operar em uma ambiente Java, ou seja, as aplicações desenvolvidas utilizando RMI (Sun) acessam métodos de objetos, desenvolvidos em Java, através da Máquina Virtual Java que deverá estar presente nas máquinas que invocam os métodos dos objetos remotos e nas máquinas que disponibilizam os objetos remotos.

Uma aplicação RMI é composta basicamente por dois tipos de programas:

1) **Servidor –** um servidor, em RMI, é um aplicativo que cria objetos remotos, produz referências a estes objetos, para torná-los acessíveis, e espera por requisições de clientes para acessar os métodos remotos destes objetos.
2) **Cliente –** um cliente, em RMI, é um aplicativo que obtém uma referência a um, ou mais, objetos remotos do servidor e invoca os métodos remotos destes objetos.

RMI providencia mecanismos que tornam possível um servidor e um cliente se comunicarem e trocarem informações. Este tipo de aplicação é chamado de

Aplicação de Objetos Distribuídos. Uma aplicação de objetos distribuídos necessita:

1) **Localizar Objetos Remotos –** uma aplicação desenvolvida em RMI pode utilizar dois mecanismos para localizar objetos remotos: uma aplicação pode registrar um objeto utilizando o *RMI Registry* (Sun) ou passar e retornar objetos remotos, por referência, como parte de sua operação normal.
2) **Comunicar com Objetos Remotos –** Os detalhes da comunicação entre objetos remotos são tratados por RMI, do ponto de vista do programador, uma comunicação remota com um objeto assemelha se a invocação padrão de um método de um objeto Java.
3) **Envio de Objetos –** RMI permite que uma aplicação possa enviar um objeto (no formato de *bytecodes*) para um objeto remoto. Os mecanismos necessários para enviar um objeto, como transmitir dados, são providenciados pelo RMI.

O recurso de envio de objeto entre máquinas virtuais permite que novos tipos sejam introduzidos em uma outra máquina virtual estendendo dinamicamente as características da aplicação sem a necessidade de recompilar a aplicação para se adicionar novas funcionalidades.

A seguir, como mostra a Figura 13, uma aplicação “Cliente” acessando objetos remotos disponibilizados pelo aplicativo “Servidor” utilizando RMI.

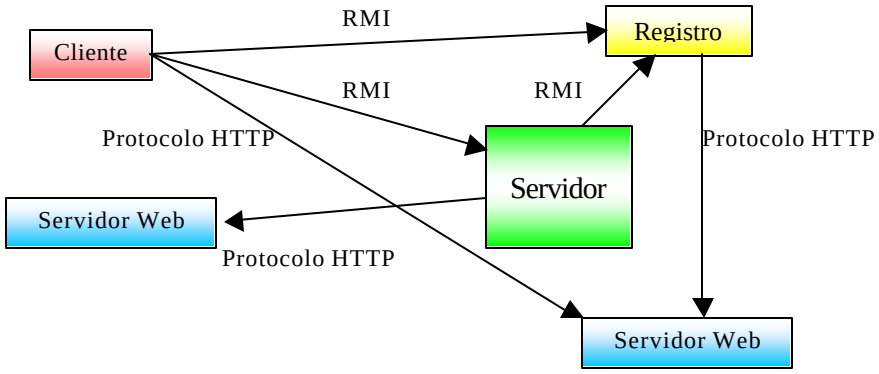

Figura 13: Um Exemplo de uma Aplicação RMI

5.4.1 Interfaces Remotas, Objetos e Métodos

Um aplicativo RMI, como qualquer outro aplicativo Java, é composto de interface e classes. As interfaces definem as assinaturas dos métodos e as classes definem as implementações dos métodos definidos pela interface.

Entretanto, estas classes podem implementar métodos que não foram definidos por uma determinada interface. Quando se desenvolvem aplicativos distribuídos, utilizando RMI, algumas dessas implementações podem residir em diferentes máquinas virtuais Java.

Objetos que possuem métodos que são chamados por outras máquinas virtuais são chamados de objetos remotos. Um objeto torna se remoto quando implementa uma interface remota, ou seja, quando este estende a classe java.rmi.Remote ou alguma classe derivada desta.

RMI trata diferentemente um objeto local de um objeto remoto, quando este é passado de uma máquina virtual para outra. Ao invés de fazer uma cópia do objeto remoto na máquina virtual que fez a invocação de um método deste, RMI passa para esta máquina virtual um *stub*. Um *stub* atua como um representante

local para um objeto remoto, basicamente este é uma referência a um objeto remoto. A aplicação local invoca um método no *stub* local, o qual é responsável por fazer a chamada ao método do objeto remoto. O *stub* implementa a mesma interface que o objeto remoto implementa.

5.5 DCOM

DCOM ou *Distributed COM*, definido pela *Microsoft Corporation*, é a extensão distribuída do COM (*Component Object Model*), onde se implementou uma chamada de procedimento de objetos remotos (*Object Remote Procedure Call* - ORPC) com o objetivo de suportar objetos remotos.

O COM define como os componentes de software e os seus clientes (outros componentes que se utilizam dos seus serviços) podem se conectar sem a necessidade de qualquer recurso intermediário. Um servidor COM pode criar instâncias de objetos de múltiplas classes. Um objeto COM pode suportar múltiplas interfaces, cada uma representando uma visão ou comportamento diferente do objeto.

Uma interface consiste de um conjunto de métodos funcionalmente relacionados. Um cliente COM interage com um objeto COM adquirindo um ponteiro para uma das interfaces do objeto e chamando métodos, como se o objeto residisse no espaço de endereço do cliente. O COM especifica que qualquer interface deve seguir um layout de memória padrão, que é o mesmo que a tabela de funções virtuais do C++.

Nos sistemas operacionais atuais, os processos são protegidos uns dos outros. Um cliente que precisa se comunicar com um componente em outro processo não pode chamar o componente diretamente, mas deve usar alguma forma de comunicação interprocessos fornecida pelo sistema operacional. O modelo COM intercepta chamadas a partir do cliente e as direciona para o componente em outro processo.

5.5.1 Independência de Localização

Com o DCOM, os detalhes sobre as questões de localização dos componentes não são especificados no código fonte, estando eles no mesmo processo, ou em uma máquina em qualquer parte do mundo.

Em todos os casos, a forma como o cliente se conecta àum componente e chama os métodos do componente é idêntica. Da mesma forma que o DCOM não requer mudanças no código fonte, também não é necessária a recompilação do mesmo. A independência de localização do DCOM simplifica a tarefa de distribuição dos componentes de uma aplicação.

5.5.2 Balanceamento de Carga Estático

É um método de balanceamento de carga que associa para certos usuários certos servidores que executam a mesma aplicação. As aplicações baseadas em DCOM podem ser facilmente configuradas para utilizar servidores específicos.

Um método flexível usa um componente de referência dedicado. Este componente reside em uma máquina servidora bem conhecida. Os componentes clientes conectam-se primeiramente com este servidor, requisitando uma referência para o serviço que eles procuram. Ao invés de só retornar o nome do servidor, o componente de referência pode realmente estabelecer uma conexão com o servidor e retorná-lo diretamente para o cliente. O DCOM então conecta-se transparentemente o cliente com o servidor.

5.5.3 Balanceamento de Carga Dinâmico

A idéia do componente referencial pode ser usada para prover um mecanismo mais eficiente de balanceamento de carga. Ao invés de basear a escolha do servidor no identificador do cliente, o componente de referência pode usar

informações sobre a carga do servidor, topologia de rede entre o cliente e os servidores disponíveis e estatísticas sobre a demanda passada de um dado usuário. A cada vez que um cliente se conecta a um componente, o componente de referência pode associá-lo para o servidor disponível mais adequado no momento.

O DCOM não especifica métodos de reconexão (nos casos de perda da conexão ou carga desigualmente distribuída), nem mesmo informações de estado sobre as conexões. Estas soluções, se necessárias, são possíveis, porém dependem de implementação adicional.

5.5.4 Tolerância a Falhas

O DCOM fornece suporte básico para tolerância a falhas no nível de protocolo. Um mecanismo de *pinging* detecta falhas de falhas de rede do lado cliente. Se a rede se restabelece antes do intervalo de *timeout* definido, o DCOM restabelece as conexões automaticamente. Uma técnica diz respeito ao componente de referência. Quando os clientes detectam a falha de um componente, eles se conectam novamente ao mesmo componente de referência que estabeleceu a primeira conexão. Neste caso, as aplicações perderão informações anteriores, bem como a sua consistência, devendo tratar isso em níveis superiores.

Outra técnica é chamada de "*hot backup*". Duas cópias do mesmo componente servidor são executadas em paralelo em diferentes máquinas, processando as mesmas informações. Os clientes podem explicitamente conectar ambas as máquinas simultaneamente.

5.5.5 Segurança

O DCOM usa uma estrutura padrão de segurança fornecida pelo Windows NT. O Windows NT fornece um conjunto de provedores de segurança internos que suportam múltiplos mecanismos de identificação e autenticação. Uma parte central

dessa estrutura de segurança é o diretório de usuários, que armazena as informações necessárias para validar as credenciais de um usuário (nome, senha, chave pública).

5.6 Conclusões do Capítulo

Neste capítulo foi possível observar que Corba RT, apesar de prover desenvolvimento de Sistemas de Tempo Real, possui algumas limitações para poder garantir o determinismo do sistema.

Já DCOM é dependente da arquitetura Microsoft, e apesar de prover serviços necessários para distribuição, sua complexidade gera sobrecarga de processamento desnecessário para um simples serviço de servidor de nomes (que será desenvolvido neste trabalho). Também, a sua dependência de plataforma limita o desenvolvimento de aplicações distribuídas sobre meios heterogêneos.

Por outro lado, RMI provê a implementação de aplicações sobre meios heterogêneos. Mas, apesar disto, a linguagem de programação fica limitada a Java.

Uma das soluções seria a implementação de um servidor de nomes utilizando a linguagem C, utilizando um protocolo próprio para cadastrar e gerenciar a localização dos processos no sistema. Tal servidor de nomes será melhor estruturado no próximo capítulo.

Capítulo 6
Integração dos Requisitos Temporais

Semelhante ao trabalho de [7], este trabalho atuará provendo garantias temporais à troca de mensagens efetuadas sobre um meio de comunicação. Para isto, o trabalho descrito neste documento atuará estendendo a comunicação interprocessos através de uma rede de comunicação. Este trabalho também proverá políticas de atendimento às mensagens de tempo real, que podem ser trocadas entre processos de uma mesma estação, ou entre estações distintas, através da rede de comunicação.

Neste Kernel serão implementadas primitivas de troca de mensagens que estenderão a funcionalidade das primitivas já existentes em outros Sistemas Operacionais. Estas primitivas possibilitarão que ocorra a passagem de parâmetros de Tempo-Real juntamente com os parâmetros da Rede de Comunicação.

Figura 14: Proposta de Arquitetura

6.1 Tarefas do Sistema Operacional

As tarefas do Sistema Operacional proposto são executadas dentro de processos de usuário. Cada um destes processos possui seu próprio escopo de variáveis e funções. As tarefas definidas durante a inicialização do sistema serão criadas como processos filhos do processo do kernel. As demais tarefas criadas a partir destes processos serão criadas como processos filhos dos processos criadores. Tal hierarquia pode ser observada na Figura 15.

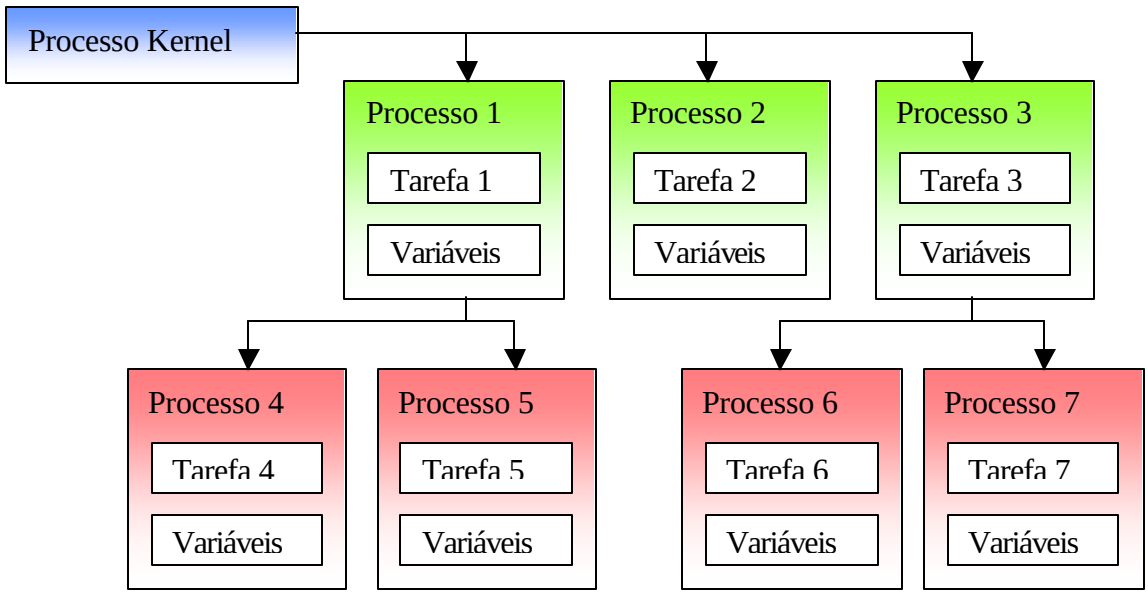

Figura 15: Hierarquia de Processos

Sendo o acesso a variáveis globais limitado ao escopo do processo, uma tarefa não pode acessar uma variável de uma outra tarefa. Uma alternativa para tal acesso é através da criação de uma memória compartilhada. Tanto a criação destas tarefas quanto a criação de tal tipo de memória serão vistos no capítulo posterior.

6.2 Funções de Troca de Mensagens de um RTOS

Como dito anteriormente no capítulo 3, as primitivas de troca de mensagens são baseadas em *Send* e *Receive*, que podem ser diretas ou indiretas, bloqueantes ou não bloqueantes. Porém, no sistema operacional proposto, estas funções são responsáveis pela garantia de entrega das mensagens com cumprimento dos requisitos temporais adquiridos do Processo. Para isto, o sistema operacional mantém os requisitos temporais do processo na instância de MessageRT, com o objetivo de controlar prazos, prioridades, latências entre outros requisitos de uma mensagem a ser enviada. Diante estas características é possível demonstrar a interface com estes métodos:

```
KernelStatus_T Kernel_MessageRTSend(MessageInstance,Message,TargetAccessPoint);
KernelStatus_T Kernel_MessageRTReceive(MessageInstance,Message,SenderAccessPoint);
```

Nota-se a importância da utilização de uma rede de comunicação previsível, já que o RTOS dependerá de taxas constantes de transmissão, e tempos de entregas fixos. Algumas tecnologias provêm estas características, como *Token Ring* ou FDDI, utilizando fichas para transmissão de dados numa topologia em anel, ou ATM, com reserva de recurso formando um circuito lógico.

Porém, mesmo que redes do tipo *Ethernet* não sejam normalmente utilizadas para aplicações de Tempo-Real devido a sua natureza não determinística, é comprovado por [8,9 e 10] que tal tipo de rede pode prover garantias temporais a aplicações de tempo real caso a rede permaneça dentro de um limite pré-determinado.

6.3 Troca de Mensagens de Tempo Real sobre uma Rede de Comunicação

Nesta seção será tratada a extensão das funções de troca de mensagens de um sistema operacional para abranger uma rede de comunicação, assim como os requisitos temporais como políticas de atendimento.

6.3.1 Tratamento da Política de Atendimento das Mensagens

Para garantir o cumprimento dos requisitos temporais na troca de mensagens, este trabalho propõe utilizar o protocolo *Immediate Priority Ceiling* para manipular a prioridade dos processos envolvidos em uma troca de mensagens.

Para isto, designaremos *priority ceilings* para as mensagens. Tal *priority ceiling*, pela definição do protocolo, receberá o maior valor entre as prioridades dos processos interlocutores. Também pela definição do protocolo, os demais processos envolvidos na mesma comunicação herdarão a prioridade do *priority ceiling*.

Deste modo, na chegada de uma mensagem com o *priority ceiling* maior do que a prioridade do processo em execução pode fazer com que o processo receptor da mensagem seja colocado em execução, mesmo que este possua uma baixa prioridade.

Utilizaremos um exemplo para melhor ilustrar esta situação: suponhamos que um processo de alta prioridade em execução envie uma mensagem de alta prioridade para um determinado processo em uma outra máquina. Suponhamos também que esta outra máquina possua um outro processo em execução, diferente daquele que irá receber a mensagem, e este processo em execução tenha prioridade menor do que a da mensagem enviada.

Segundo o protocolo de Immediate Priority Ceiling empregado neste trabalho, mesmo que o processo receptor da mensagem seja menos prioritário do que o em execução, o processo receptor herdará a prioridade da mensagem e, caso se tornar mais prioritário após a herança, ele será colocado em execução na máquina receptora, como exibido na Figura 16.

A notação de valores de prioridades utilizada no sistema operacional proposto é que quanto menor o valor da prioridade mais critica é a tarefa. Deste modo, podemos formalizar os passos para esta transmissão através de um algoritmo. Tal algoritmo para o envio de mensagem pode ser descrito no procedimento descrito pela Figura 17.

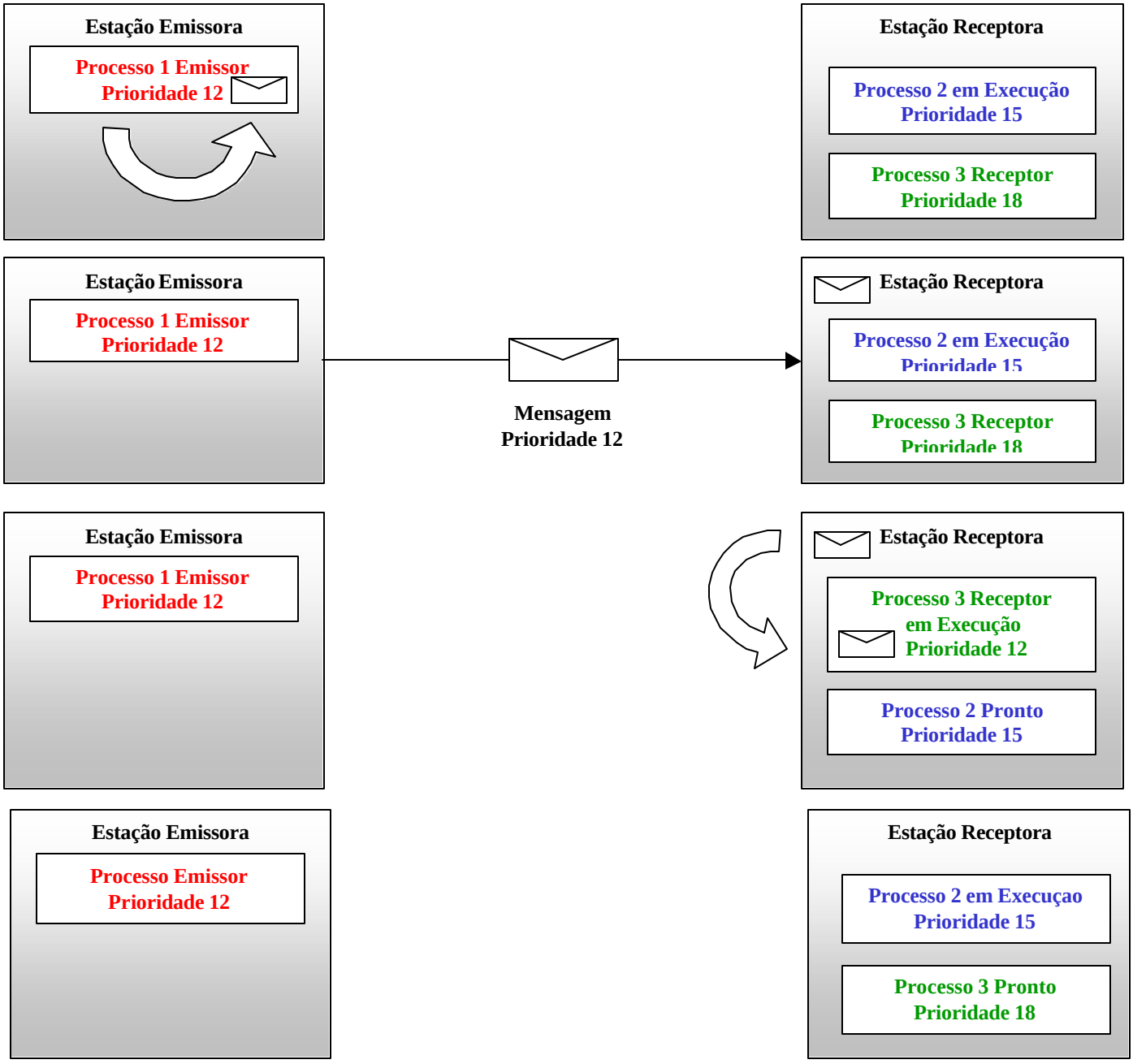

Figura 16: Troca de Mensagem com Prioridade através da Rede

Semelhante ao trabalho de [4], um processo de baixa prioridade, receptor de uma mensagem de alta prioridade herdaria a prioridade do processo emissor. Esta abordagem possibilita maior transparência ao programador da Aplicação, sendo

que a primitiva de *SendMessage* utiliza os valores já determinados na estrutura de lista de processos.

Estação Emissora	
1	Processo Emissor faz uma chamada a Kernel_MessageRTSend().
2	O GlobalAccessPoint passado como parâmetro a Kernel_MessageRTSend() é resolvido em um endereço.
3	Kernel_MessageRTSend() cria o cabeçalho da mensagem contendo o Priority Ceiling e o Endereço do Processo Receptor.
4	A mensagem criada no passo anterior é enviada pela rede.
5	Processo Emissor volta àsua execução normal
Estação Receptora	
6	*Processo Receptor chama Kernel_MessageRTReceive() e permanece bloqueado até o recebimento da mensagem.*
7	Ao ocorrer o recebimento da mensagem, caso o Priotiy Ceiling da mensagem seja maior que a prioridade dinâmica do processo receptor, este processo herdará a prioridade da mensagem e pode ser colocado em execução, caso ele se torne mais prioritário que o processo em execução.

Figura 17: Procedimento para Transmissão de Mensagens

O trabalho de [5] propõe que o tráfego de chegada de dados em uma estação seja escalonado com a mesma prioridade do processo que recebe o tráfego. Por outro lado, o sistema operacional proposto analisa todas as mensagens recebidas, e sua entrega para um determinado o processo receptor é feita somente dado um dos casos abaixo:

- Se o processo receptor está em execução;
- Se o após herdar o Priotity Ceiling da mensagem, o processo receptor passar a ser mais prioritário que o processo em execução e por este motivo ganhar uma fatia de execução;
- Caso contrário, o processo somente receberá a mensagem quando entrar em execução.

Observa-se também que este tipo de política de atendimento afetará também o modo com que é feito o escalonamento de processos no sistema. Esta implementação, além de tratar os requisitos temporais da troca de mensagens, possibilitará uma solução para o problema da inversão de prioridade [14].

6.3.2 Manipuladores de Recepção e Envio de Mensagens

Devido à necessidade de transmissão da Prioridade, cujo valor está agregado ao conteúdo da mensagem, é necessário criar um conjunto de regras para inserir e extrair estes campos de dentro de uma mensagem.

Além destes campos referentes aos Requisitos de Tempo Real, também deve ser enviada a referência do processo alvo que irá receber a mensagem. Estes campos podem ser referenciados através de uma PDU de Aplicação, aplicada diretamente sobre a camada de transporte. Tal PDU é definida pela Figura 18 e possui o delimitador "|" entre os campos.

Tipo da Mensagem	Endereço de Resposta	Porta de Resposta
Ponto de Acesso Global	Endereço da Estação	Porta da Estação
Mensagem		

Figura 18: PDU Troca de Mensagens

O Campo Tipo da Mensagem pode assumir os seguintes valores: `GAP_ADD_REQUEST`, `GAP_GET_REQUEST`, `GAP_REMOVE_REQUEST`, `GAP_UPDATE_REQUEST` e `GAP_FINISH_REQUEST`.

Os campos Endereço de Resposta e Porta de Resposta possuem respectivamente o endereço IP e Porta referentes a comunicação UDP utilizada para troca de mensagens.

O Ponto de Acesso Global é um valor inteiro e único utilizado para identificar um interlocutor de uma comunicação. Os campos Endereço da Estação e Porta da Estação são respectivamente o endereço IP e Porta associados ao Ponto de Acesso Global.

O Campo Mensagem contém efetivamente a mensagem, composta por uma cadeia de caracteres, a ser transmitida pela rede.

6.4 Alocação de Nomes

Por este sistema trabalhar com troca de mensagens de processos distribuídos, é interessante que o sistema suporte a independência da localização dos processos. Tal independência proveria ao desenvolvedor de STRs maior flexibilidade e transparência na localização dos processos dentro do sistema.

Para que isto ocorra, podemos utilizar 2 técnicas: utilizar nomes globais para os processos, assim como é feito no Sistema Operacional Virtuoso; ou utilizar um Servidor de Nomes para registrar e identificar a distribuição dos processos.

6.4.1 Nomes Globais de Processos

Utilizar nomes de processos que não se repetem em todo o sistema é comumente utilizado na implementação de RTOSs. Um exemplo de RTOS que utiliza este tipo de implementação é o Virtuoso. Uma das maiores vantagens é a facilidade de implementação o alto desempenho, já que esta técnica dispensa a utilização de um servidor de nomes. Como desvantagem, o processo deverá terminar sua execução na mesma estação que ocorreu a sua criação.

Para a utilização de nomes globais não é necessária a implementação de um servidor de nomes para o registro das tarefas. Os nomes são ligados as tarefas em tempo de projeto, onde cada nome é mapeado para apenas uma tarefa, não sendo aceitável repetir o mesmo nome para duas tarefas distintas dentro do sistema.

6.4.2 Servidor de Nomes

Para que um processo esteja apto a receber mensagens de processos localizados em outras estações de trabalho, ele deve se cadastrar em um Servidor de Nomes. Este Servidor de Nomes resolve o valor de Ponto de Acesso Global em endereço IP e Porta, e a interação com os outros elementos de rede pode ser exemplificada nas Figuras 19 e 20.

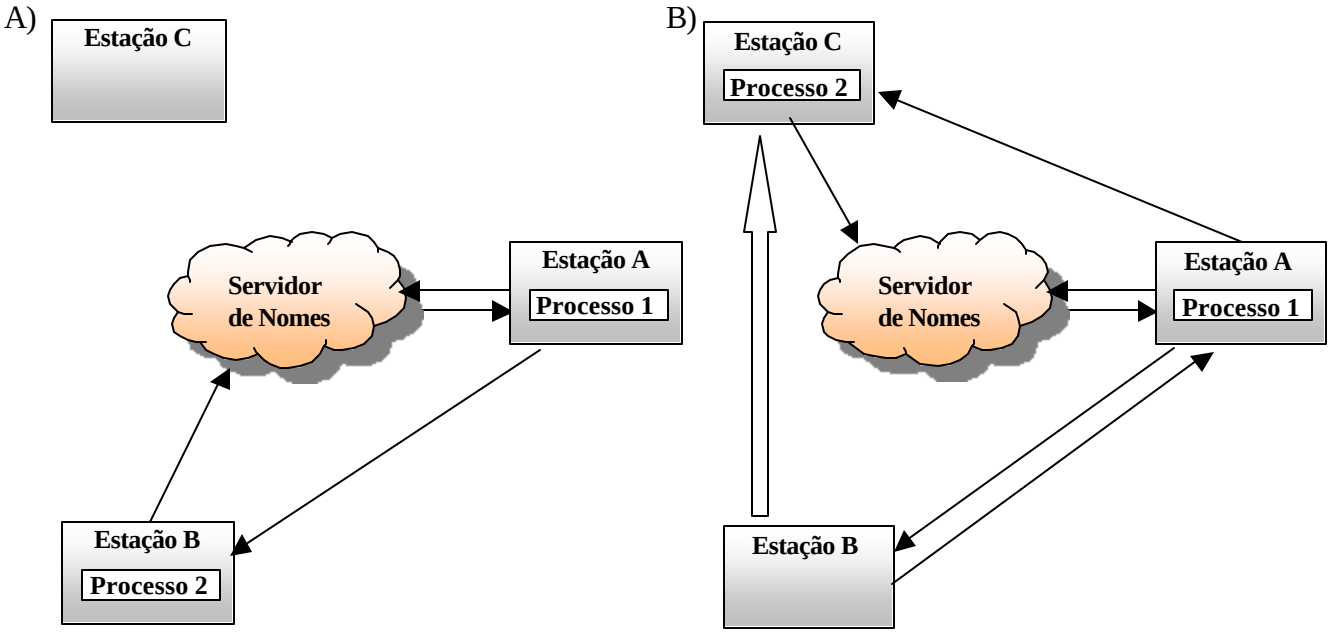

Figura 19: Caso de Uso; A) Antes da Migração do Serviço; B) Após a Migração do Serviço

O servidor de nomes gerencia o cadastro de Pontos de Acesso Global (GAP) para os processos que utilizarão troca de mensagens em estações distintas. As funcionalidades pelas quais o Servidor de Nomes se torna responsável são Inclusão, Consulta, Atualização e Remoção de Registros.

Com o intuito de ocorrer tal cadastro, é necessário fornecer para o Servidor de Nomes o Ponto de Acesso Global, assim como o endereço IP e o número da porta de acesso (Port) da estação que está requerendo o cadastro. Sempre que uma aplicação desejar chavear a localidade, isto é, estações, onde os processos estão

situados, é necessário que ele informe sua localização atual para o Servidor de Nomes de modo a mantê-lo sempre atualizado.

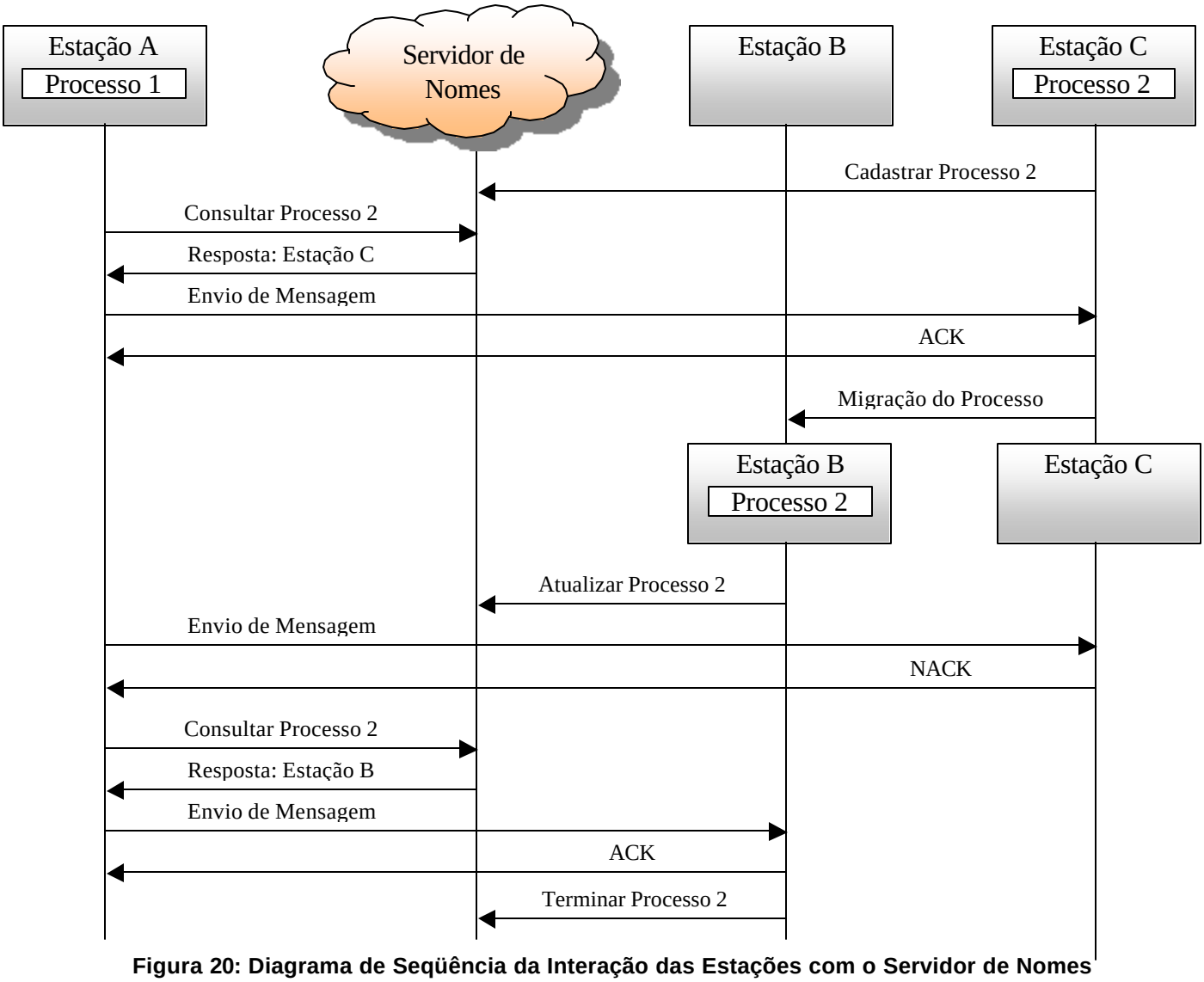

Figura 20: Diagrama de Seqüência da Interação das Estações com o Servidor de Nomes

Quando ocorrer a atualização ou término da execução de um processo cadastrado no Servidor de Nomes, o sistema operacional da estação que o estiver hospedando se incumbirá de informar ao Servidor de Nomes. O Servidor de Nomes, por sua vez, irá descadastrá-lo de sua tabela.

Futuras tentativas de envio de mensagens a uma estação que não esteja mais em uso causará uma falha na busca no cache (*cache fault)* na estação emissora. Tal *cache fault* ocasionará uma nova consulta ao servidor de nomes.

6.4.3 Cache de Ponto de Acesso Globais

Para que um processo envie uma mensagem para um outro processo, sendo que o processo receptor está localizado em uma estação distinta do emissor, é necessário resolver a identificação global do processo receptor em um endereço IP e uma Porta. Uma vez que tenha ocorrido esta consulta no servidor de nomes, tais dados podem permanecer armazenados em um Cache, assim como exemplificados na Figura 21.

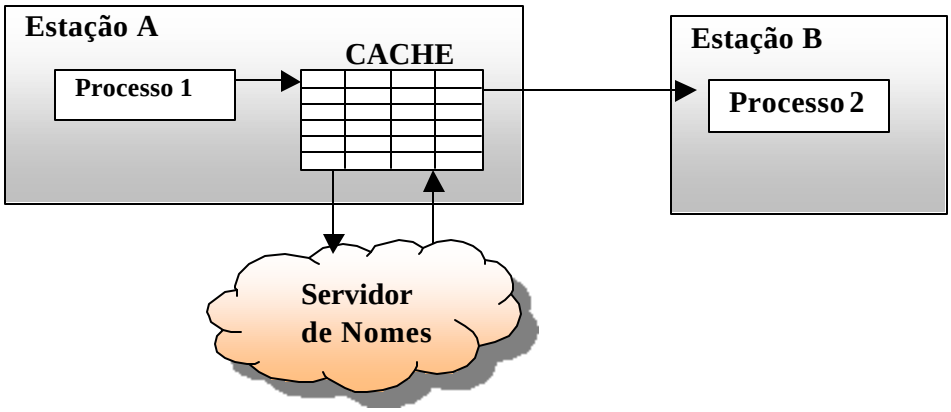

Figura 21: Tabela Local de Pontos de Acesso Global

Tal Cache deve armazenar o Ponto de Acesso Global, o Endereço IP e a Porta do Processo Receptor. Esta tabela tem por finalidade reduzir a quantidade de troca de mensagens pela rede de comunicação para comunicação com o Servidor de Nomes, uma vez que ela é utilizada como um *cache* das informações recebidas do Servidor de Nomes.

Uma vez que ocorra uma falha de transmissão referente a uma entrada consultada nesta tabela, a entrada da Tabela Local que produziu a falha deve ser atualizada com novas informações vindas do Servidor de Nomes. Caso o Servidor de Nomes retorne a informação que o processo consultado foi terminado, a entrada será removida do Cache.

Já, ao utilizar a idéia de se manter um Cache para tentar diminuir uma sobrecarga no meio de comunicação, artifícios para manter a consistência entre as tabelas devem ser levantados. O trabalho de [6] trata justamente isto. Entre os métodos abordados por ele, são avaliados:

- *Poolling Every Time*, onde consultas a cada estação são feitas para avaliar se o Tempo de Vida expirou. Esta técnica ocupa muito recurso do meio de comunicação durante as trocas de mensagens.
- *Invalidation*, onde um componente mantém a rastreabilidade de quais tabelas contêm um determinado elemento, invalidando apenas os elementos das tabelas necessárias. Porém, este método ocupa recursos de processamento de um terceiro componente.
- *Adaptative Time to Live*, onde o Tempo de Vida de uma determinada entrada da tabela varia segundo um comportamento probabilístico, deste modo, ocupando recursos do sistema.

Porém, o sistema operacional proposto possui um método mais simples e eficiente para impedir que as informações localizadas no Cache percam a validade e permaneçam lá sem uso por um tempo indefinido. No instante que ocorrer um acesso a uma entrada da tabela cujo tempo de vida já expirou, tal entrada é removida da tabela e uma nova consulta é feita ao Servidor de Nomes.

6.4.4 PreFetching

Com o intuito de minimizar a taxa de *cache faults* nos instantes iniciais da execução do sistema, é possível que ocorra uma fase de *prefetching*. Tal fase permitiria a busca dos identificadores globais no Servidor de Nomes durante a inicialização do sistema. Tais nomes são providos pelo desenvolvedor da aplicação antes mesmo de dar início àexecução do kernel.

6.5 Conclusões do Capítulo

Através deste capítulo, é possível observar que o Middleware e o sistema operacional acima descrito provêem troca de mensagens utilizando requisitos temporais em todas mensagens transmitidas pela rede de comunicação. E tais requisitos, transmitidos na estrutura de dados de uma mensagem, permitem ao Sistema Operacional mantenha processos interlocutores se comunicando com a mesma prioridade.

Capítulo 7
Estudo de Caso

Como estudo de caso, tal sistema tem sua implementação para PCs, tendo o SãoCarlOS trabalhando sobre Linux e utilizando UDP/IP, fazendo chamadas do tipo *socket*. Uma estrutura de tal arquitetura pode ser observada na Figura 22, onde as aplicações podem fazer chamadas a métodos nativos do SãoCarlOS.

O SaoCarlOS é um Sistema Operacional de Tempo-Real que provê suporte à Troca de Mensagens de Tempo-Real através da implementação do protocolo Immediate Priority Ceiling e suporte a Nomes Globais, Servidor de Nomes, Cache e Prefetching de modo a prover independência de localização.

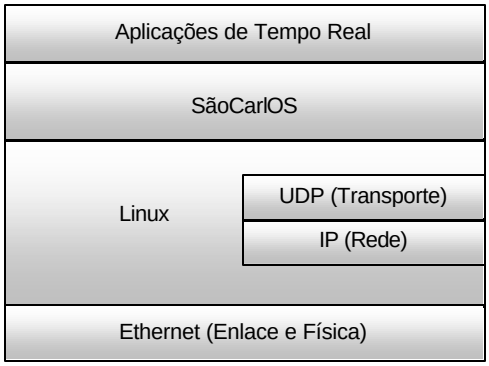

Figura 22: Estudo de Caso sobre UDP/IP

A arquitetura do SãoCarlOS pode-se dizer semelhante à arquitetura proposta por [1], onde existem *Threads* de usuário que são controladas por um Kernel de Tempo Real executando sobre um outro Kernel. Neste trabalho processos pertencentes ao usuário são escalonados por um Kernel executando sobre a plataforma Linux.

Uma implementação de troca de mensagens sobre diferentes tipos de primitivas disponibiliza ao projetista uma diversidade maior de chamadas de sistema, provendo maior adequação à aplicação de tempo real. Neste caso, as arquiteturas propostas neste trabalho provêem primitivas distintas para troca de Mensagens

RT, as quais permitem que a aplicação escolha entre troca de mensagens com garantias temporais e troca de mensagens sem garantias temporais utilizando apenas UDP/IP.

A interface de programação do SaoCarlOS é baseada em chamadas de sistema. Este capítulo aborda cada chamada de sistema do kernel SaoCarlOS e provê exemplos de utilização destas chamadas.

7.1 Criação de Tarefas

No kernel SaoCarlOS, uma tarefa (ou *task*) é implementada dentro de uma função, e esta função é passada como parâmetro a uma chamada de sistema que cria a instância da tarefa. Um exemplo para a criação de uma tarefa pode ser observado na Figura 23.

```
void
Task_1_Implementation()
{
  int i=20000;
  while(--i)
  {
        printf("\n\tTask1");
        fflush(stdout);
  }
}
```

Figura 23: Implementação de uma Tarefa

Assim como mencionado no Capítulo 3 – Sistemas Operacionais, uma tarefa pode ser criada de forma estática ou dinâmica. Ambas implementações são suportadas pelo kernel SaoCarlOS e são descritas nas subsecções a seguir.

7.1.1 Criação de Tarefas Estáticas

Uma tarefa criada estaticamente é aquela definida em tempo de projeto e cuja execução é iniciada durante a fase de inicialização do sistema. Para criar uma tarefa estática no SaoCarlOS, é necessário que ocorra a declaração, alocação de

memória e inicialização da variável *Priority* da estrutura Process_T. Também é necessário que ocorra uma chamada para Kernel_KernelInit() que inicializará todas as estruturas internas do kernel. O cadastro da função que contém a implementação da tarefa é feita pela função Kernel_ProcessCreate, passando como parâmetro a estrutura do tipo Process_T e a implementação da tarefa. Um exemplo para esta criação pode ser visto na Figura 24.

A tarefa criada estaticamente é obtida a partir de um fork() a partir do processo do kernel durante a inicialização do sistema. A execução desta tarefa é feita através da execução do ponteiro para função passado como argumento da chamada de sistema. Durante esta fase todas as tarefas criadas são interrompidas por um SIGSTOP e colocadas na lista de Pronto (READY).

```
int
main()
{
   Process_T *Process = (Process_T *)
      Kernel_MemoryAllocate(sizeof(Process_T));

   Kernel_KernelInit();

   Process->Priority = 10;

   Kernel_ProcessCreate(Process, Task_0_Implementation);

   Kernel_KernelStart();
}
```

Figura 24: Criação de uma Tarefa Estática

7.1.2 Criação de Tarefas Dinâmicas

Uma tarefa criada dinamicamente é aquela que acontece em tempo de execução e a tarefa é iniciada após o inicio da execução do Kernel_KernelStart(). E para criar uma tarefa estática no SaoCarlOS basta passar os parâmetros KERNEL_NULL e a implementação da tarefa para a chamada de sistema Kernel_ProcessCreate(). A Figura 25 demonstra como criar uma tarefa dinâmica.

Analogamente a criação de tarefas estáticas, as tarefas dinâmicas também são criadas com o uso de fork() a partir do processo de usuário e executando o ponteiro para função passado como argumento da chamada de sistema. Porém após a criação, o processo pai (criador do novo processo), envia um evento do SãoCarlOS ao processo kernel, informando a criação da nova tarefa. Este por sua vez, atualiza a estrutura de processos do SãoCarlOS com o novo processo herdando todos os requisitos temporais do processo criador.

```
void
Task_0_Implementation()
{
  Kernel_ProcessCreate(KERNEL_NULL, Task_1_Implementation);
}
```

Figura 25: Criação de uma Tarefa Dinâmica

7.2 Prioridades de Tarefas

As prioridades de tarefas no kernel SaoCarlOS são definidas como prioridades estáticas ou dinâmicas. As prioridades estáticas são aquelas que são inicializadas durante a criação de um processo estático e permanecem com o mesmo valor durante toda a execução do sistema. Os processos criados dinamicamente herdarão a prioridade de seus processos criadores. Já as prioridades dinâmicas podem ser modificadas em tempo de execução através de primitivas para mudança de prioridades e troca de mensagens de tempo real. Esta segunda forma será vista em uma seção mais adiante.

7.2.1 Prioridades Estáticas de Tarefas

Como no kernel SaoCarlOS os processos são executados por prioridade, vale lembrar que todos os processos de mais alta prioridade são executados antes que um de prioridade mais baixa seja executado. Desta forma, para processos com mesma prioridade, o SaoCarlOS tem um comportamento semelhante ao algoritmo Round Robin.

```
int
main()
{
   int i = 0;
   Process_T *Process[6];

   for(i=0;i<6;i++)
      Process[i] = (Process_T *)
         Kernel_MemoryAllocate(sizeof(Process_T));

   Kernel_KernelInit();

   Process[0]->Priority = 1;
   Process[1]->Priority = 5;
   Process[2]->Priority = 10;

   Kernel_ProcessCreate(Process[0], Task_1_Implementation);
   Kernel_ProcessCreate(Process[1], Task_2_Implementation);
   Kernel_ProcessCreate(Process[2], Task_3_Implementation);

   Kernel_KernelStart();
}
```

Figura 26: Atribuição de Prioridades Estáticas à Tarefas

A prioridade estática é definida pela estrutura do processo Process_T em Priority, que define a prioridade estática do processo. A prioridade estática deve ser definida antes da chamada de sistema Kernel_ProcessCreate();. Um exemplo de atribuição de prioridades estáticas para processos pode ser observado na Figura 26.

7.2.2 Prioridades Dinâmicas de Tarefas

Já a prioridade dinâmica é determinada pela variável DynamicPriority localizada dentro da estrutura Process_T. Porém é desaconselhável alterar o valor de DynamicPriority em tempo de execução, pois isto pode causar inconsistência desta variável no contexto do kernel e no contexto da tarefa.

Deste modo, para modificar a prioridade dinâmica de um processo pode-se utilizar o método `Kernel_ProcessSetPriority()`. Como parâmetros para este método são passadas a identificação do processo e a prioridade a ser atribuída àquele processo.

De forma a melhor exemplificar este processo, observe as duas chamadas aos métodos `Kernel_ProcessSetPriority()` localizados na Figura 27. A primeira chamada envia para o kernel a requisição de modificação da prioridade do próprio processo. Já a segunda chamada faz a requisição de mudança da prioridade do processo que criou dinamicamente o processo em execução.

```
void
Task_1_Implementation()
{
  Kernel_ProcessSetPriority(Kernel_ProcessGetProcessID(),2);
  Kernel_ProcessSetPriority(Kernel_ProcessGetParentProcessID(),2);
}
```

Figura 27: Atribuição de Prioridades Dinâmica à Tarefas

7.3 Bloqueio de Tarefas

As tarefas do kernel SaoCarlOS podem atingir o estado Bloqueado através de três maneiras:

A forma mais comum ocorre quando o processo espera a liberação de um determinado recurso, seja ele a leitura de um socket, pipe ou a espera de um evento de tempo. O bloqueio a partir destas funcionalidades será discutido a seguir.

Um outro modo é através do uso da chamada Kernel_ProcessWait(), causando à tarefa que executou a chamada de sistema ser Bloqueada.

O último método é através da chamada Kernel_ProcessWaitProcess(), passando como parâmetro o ProcessID do processo alvo. Esta chamada irá causar o bloqueio do processo alvo, cuja identificação foi passada como parâmetro ao método.

Exemplos de utilização para os dois últimos métodos acima descritos podem ser encontradas na Figura 28. A primeira chamada desta Figura causará o bloqueio do processo Pai do processo que executou a função e a segunda chamada causará o bloqueio do processo em execução.

```
void
Task_1_Implementation()
{
  Kernel_ProcessWaitProcess(Kernel_ProcessGetParentProcessID());
  Kernel_ProcessWait();
}
```

Figura 28: Bloqueio de Tarefas do SaoCarlOS

7.4 Desbloqueio de Tarefas

O desbloqueio de tarefas no kernel SaoCarlOS, pode ocorrer em duas situações:

Estando um processo bloqueado à espera de um evento, como descrito na primeira situação da seção anterior, e dado o acontecimento deste evento, o processo será considerado pronto a executar, entrando assim na lista de Pronto.

A segunda situação é através da chamada de sistema Kernel_ProcessResume(), passando como parâmetro a identificação do processo a ser desbloqueado. Neste caso, caso o processo a ser desbloqueado possua prioridade superior à prioridade do processo em execução, o escalonador se incumbirá de causar uma preempção e o processo recém acordado receberá sua fatia de tempo de execução.

Diferentemente da seção anterior, não existe um modo do processo executar uma chamada de sistema para desbloquear o próprio processo, pois estando ele bloqueado, não é possível que ele faça alguma chamada. Um exemplo para a segunda situação estudada nesta seção pode ser observada na Figura 29.

```
void
Task_1_Implementation()
{
  Kernel_ProcessResume(Kernel_ProcessGetParentProcessID());
}
```

Figura 29: Desbloqueio de Processos no SaoCarlOS

7.5 Liberação do Tempo de Execução (Yield)

O kernel SaoCarlOS também disponibiliza ao programador de Aplicações a liberação do processador por parte de um processo. Para isto basta que ocorra a chamada ao método Kernel_ProcessYield(); e o processo em execução voltará a lista de prontos.

Caso não exista nenhum outro processo com a mesma (ou maior) prioridade a ser executado, o processo que fez a chamada ao método Kernel_ProcessYield(); ganha novamente o seu tempo de execução.

Este método pode ser melhor exemplificado através da Figura 30.

```
void
Task_1_Implementation()
{
  Kernel_ProcessYield();
}
```

Figura 30: Liberação do Tempo de Execução (Yield)

7.6 Finalização de Tarefas

Tarefas do Kernel SaoCarlOS podem ser finalizadas em quatro situações.

A primeira situação é o término da execução da tarefa, após toda sua execução. Neste caso, eventos são trocados internamente no kernel identificando que a tarefa terminou.

A segunda situação é gerada a partir da finalização da execução do kernel. Este, por sua vez, atribui o estado Terminado ao processo, e durante o próximo escalonamento, este será removido das estruturas internas do kernel.

Mas assim, analogamente como demonstrado anteriormente, é possível forçar a finalização de uma tarefa através da chamada Kernel_ProcessTerminate();. Ao utilizar esta chamada, a tarefa chamadora é suspensa, seu estado é transicionado para Terminado, e durante a próxima execução do kernel, a estrutura de processos será atualizada, removendo este processo.

Finalmente, também é possível finalizar a execução de um outro processo utilizando a chamada Kernel_ProcessTerminateProcess(); e passando como parâmetro o ProcessID do processo alvo a ser terminado. As duas últimas situações podem ser observadas na Figura 31.

```
void
Task_1_Implementation()
{
Kernel_ProcessTerminateProcess(Kernel_ProcessGetParentProcessID());
Kernel_ProcessTerminate ();
}
```

Figura 31: Término de Tarefas no SaoCarlOS

7.7 Temporização

A Temporização no kernel SaoCarlOS é utilizada para manter uma determinada tarefa bloqueada por um determinado limite de tempo (ou até que um outro processo o acorde). A chamada de sistema que provê esta funcionalidade é a Kernel_TimeSleep() e o parâmetro passado é o valor de tempo em segundos que o processo deve ficar bloqueado. Um exemplo desta chamada de sistema pode ser observado na Figura 32.

```
void
Task_1_Implementation()
{
  int i=3;
  while(--i)
  {
        printf("\nTask1");
        fflush(stdout);
        Kernel_TimeSleep(1);
  }
}
```

Figura 32: Bloqueio de uma Tarefa por um Temporizador

7.8 Semáforos

Como dito anteriormente no Capítulo 3 – Sistemas Operacionais, um dos usos de um semáforo é para garantir exclusão mútua entre processos, de modo que não ocorra reentrância de código durante a execução de uma região crítica.

No SaoCarlOS a declaração dos semáforos utilizados durante a execução do kernel é feita durante a inicialização do sistema através da chamada de sistema Kernel_SemaphoreCreate(), e passando como parâmetro a instância alocada do tipo SemaphoreT com visibilidade global e o número total de semáforos.

Já, para inicializar cada semáforo a primitiva Kernel_SemaphoreInit é utilizada. Seu primeiro parâmetro é a Instância semáforo, seguido pelo índice do semáforo a ser inicializado, e seu terceiro parâmetro indica o valor em que o semáforo deve ser inicializado. As chamadas de sistema para criação de semáforos e inicialização dos mesmos se encontram na Figura 33.

```
SemaphoreT *SemaphoreInstance;

int
main()
{
   Process_T *Process = (Process_T *)
      Kernel_MemoryAllocate(sizeof(Process_T));
   SemaphoreInstance = (Semaphore_T *)
      Kernel_MemoryAllocate(sizeof(Semaphore_T));

   Kernel_KernelInit();
   Process->Priority = 1;

   Kernel_SemaphoreCreate(SemaphoreInstance,1);
   Kernel_SemaphoreInit(SemaphoreInstance,0,1);

   Kernel_ProcessCreate(Process, Task_1_Implementation);
   Kernel_KernelStart();
}
```

Figura 33: Criação e Inicialização de Semaforos

Analogamente à estrutura de inicialização dos semáforos, os parâmetros das primitivas Kernel_SemaphoreUp e Kernel_SemaphoreDown são também representados pela Instância do Semáforo, pelo índice do semáforo e o valor que deve ser acrescido ou decrescido no semáforo. Um exemplo da implementação de uma tarefa utilizando estas primitivas pode ser observada na Figura 34.

```
void
Task_1_Implementation()
{
   Kernel_SemaphoreDown(SemaphoreInstance,0,1);
   /* Região Crítica */
   Kernel_SemaphoreUp(SemaphoreInstance,0,1);
}
```

Figura 34: Uso das Primitivas Up e Down

Vale observar que o processo permanece bloqueado durante o período que ele está à espera da liberação de um semáforo. Isto ocorre pois internamente à função Kernel_SemaphoreDown() um evento é enviado ao processo do kernel informando que o processo irá ficar bloqueado por um semáforo. Então é feita uma chamada semop(), e após a liberação da execução pela chamada de semop(), um outro evento é enviado ao kernel informando que o processo está no estado Pronto (READY).

7.9 Canal (Pipe)

De forma a permitir a integração do kernel SaoCarlOS a outros módulos e componentes foi disponibilizado ao uso de Canais (Pipes), permitindo comunicação em duas vias (transmissão e recepção). Do mesmo modo que a implementação de semáforos, para se utilizar o Pipe também é necessário uma inicialização prévia, antes mesmo que ocorra a chamada de Kernel_KernelStart();.

```
Pipe_T Pipe;
int main()
{
   Process_T *Process_1 = (Process_T *)
      Kernel_MemoryAllocate(sizeof(Process_T));
   Process_T *Process_2 = (Process_T *)
      Kernel_MemoryAllocate(sizeof(Process_T));

   Pipe = (Pipe_T *) Kernel_MemoryAllocate(sizeof(Pipe_T));

   Kernel_KernelInit();
   Kernel_PipeInit(Pipe);

   Process_1->Priority = 1;
   Process_2->Priority = 2;

   Kernel_ProcessCreate(Process_1, Task_1_Implementation);
   Kernel_ProcessCreate(Process_2, Task_2_Implementation);

   Kernel_KernelStart();
}
```

Figura 35: Inicialização de um Canal (Pipe)

Assim como é observado na Figura 35, o Pipe deve ser um ponteiro alocado para Pipe_T e de escopo global. Para fazer a inicialização deste Canal, basta ocorrer a chamada Kernel_PipeInit(); passando como parâmetro o Pipe declarado e alocado anteriormente.

```
void
Task_1_Implementation()
{
   Kernel_PipeWrite(Pipe,"TestPipe");
}
```

Figura 36: Escrita no Canal de Comunicação

O uso do Canal também ocorre de uma forma simples. De forma a escrever no Canal, basta utilizar a chamada Kernel_PipeWrite(), passando o Canal e a cadeia

de caracteres a ser escrita no mesmo. Um exemplo de escrita no Pipe pode ser observado na Figura 36.

```
void
Task_2_Implementation()
{
   char* Buffer= (char*)
      Kernel_MemoryAllocate(PIPE_MAXIMUM_BUFFER_SIZE);

   Kernel_PipeRead(Pipe,Buffer);
   printf("\nPipe Buffer: %s",Buffer);
   fflush(stdout);
}
```

Figura 37: Leitura do Canal de Comunicação

Analogamente às funções de escrita no Canal, é possível ler do canal através da chamada de sistema encarregada pela leitura do Canal, Kernel_PipeRead(), passando como parâmetro o Canal e uma região de memória alocada para armazenar o conteúdo lido do canal, assim como observado na Figura 37.

Vale observar que tal chamada de leitura permanece bloqueada até o instante do recebimento de dados no canal. Isto ocorre, pois internamente à função Kernel_PipeRead() quando o buffer do canal está vazio, um evento é enviado ao processo do kernel informando que o processo irá ficar bloqueado pelo canal. Então é feita uma chamada read(), e após a liberação da execução pela chamada de read(), um outro evento é enviado ao kernel informando que o processo está no estado Pronto (READY).

7.10 Rede de Comunicação

Para a utilização de UDP no SaoCarlOS é necessário alocar memória para a estrutura NetworkUDP_T e a criação da instância de rede utilizando a chamada de sistema KernelNetworkUDPCreate() e passando como parâmetro a memória alocada anteriormente e o número da Porta que deve ser utilizada para a Comunicação. No caso de ser passado o valor KERNEL_NULL como porta, o Sistema Operacional escolherá aleatoriamente uma porta livre para utilizar no lugar. Tal implementação pode ser observada nas Figuras 38 e 39.

De modo a receber e ler um dado da rede de comunicação, é necessário chamar Kernel_NetworkUDPReceive(), passando como parâmetro a instância de NetworkUDP e uma região de memória pré alocada para o recebimento de tal mensagem. O recebimento de uma mensagem pela rede de comunicação pode ser observado na Figura 38.

Vale lembrar que tal primitiva mantém o processo bloqueado até o recebimento de dados pelo socket. Isto ocorre, pois internamente à função Kernel_NetworkUDPReceive(), quando o buffer do socket está vazio, um evento é enviado ao processo do kernel informando que o processo irá ficar bloqueado pelo socket. Então é feita uma chamada recvfrom(), e após a liberação da execução pela chamada de recvfrom(), um outro evento é enviado ao kernel informando que o processo está no estado Pronto (READY).

```
void
Task_1_Implementation()
{
   NetworkUDP_T* Instance =
      Kernel_MemoryAllocate(sizeof(NetworkUDP_T));
   char* Buffer = Kernel_MemoryAllocate(NETWORK_MSG_BUFFER_SIZE);

   Kernel_NetworkUDPCreate(Instance,3009);
   Kernel_NetworkUDPReceive(Instance,Buffer);
   printf("\nBuffer: %s",Buffer);
   fflush(stdout);
   Kernel_NetworkUDPTerminate(Instance);

}
```

Figura 38: Leitura de uma Instancia NetworkUDP

Já para que ocorra o envio de mensagens através da rede de comunicação, basta chamar a primitiva Kernel_NetworkUDPSend() , passando como parâmetro a instancia de NetworkUDP e o dado a ser transmitido na rede, assim como observado na Figura 39.

```
void
Task_2_Implementation()
{
   NetworkUDP_T* Instance =
      Kernel_MemoryAllocate(sizeof(NetworkUDP_T));

   Kernel_NetworkUDPCreate(Instance,KERNEL_NULL);
   Kernel_NetworkUDPSetRemote(Instance,"127.0.0.1",3009);
   Kernel_NetworkUDPSend(Instance,"Hello World.");
   Kernel_NetworkUDPTerminate(Instance);
}
```

Figura 39: Escrita em uma Instancia NetworkUDP

Após terminado o uso da rede de comunicação e assim como observado nas Figuras 38 e 39, a primitiva Kernel_NetworkTerminate() deve ser chamada para liberação do socket designado em Kernel_NetworkUDPCreate().

7.11 Mensagens de Tempo-Real

A troca de mensagens no kernel SaoCarlOS é feita através das primitivas `Kernel_MessageRTSend()` e `Kernel_MessageRTReceive()` ambas vistas nas próximas seções. Mas para fazer uso destas primitivas é necessário alocar uma instância de `RTMessage_T` para a troca de mensagens.

Também é necessário que ocorra a chamada para `Kernel_MessageRTInit()` antes de iniciar a comunicação e `Kernel_MessageRTFinish()` após finaliza-la. A identificação para o Ponto de Acesso a ser utilizado também deve ser passada como segundo parâmetro da primitiva `Kernel_MessageRTInit()`, de modo que este ponto de acesso seja registrado no Servidor de Nomes. Um exemplo de inicialização e finalização da instância de troca de mensagens pode ser observado nas Figuras 40 e 41.

7.11.1 Envio de Mensagens de Tempo-Real

Como dito anteriormente, o envio de uma mensagem de tempo real é feito através da primitiva `Kernel_MessageRTSend()`. Nela são passados como parâmetros a instancia da mensagem, a cadeia de caracteres a ser enviada e o ponto de acesso destino.

```
void
Task_1_Implementation()
{
      RTMessage_T* MessageInstance = (RTMessage_T*)
            Kernel_MemoryAllocate(sizeof(RTMessage_T));

      Kernel_MessageRTInit(MessageInstance,1);
      Kernel_MessageRTSend(MessageInstance,"Hello World",2);

      Kernel_MessageRTFinish(MessageInstance);
}
```

Figura 40: Exemplo de Envio de Mensagens de Tempo Real

Note que, assim como descrito anteriormente, o *priority ceiling* também é enviado através da mensagem. Este *priority ceiling* irá atuar diretamente na prioridade dinâmica do processo receptor da mensagem. Um exemplo de envio de mensagens de tempo real pode ser observado na Figura 40.

7.11.2 Recebimento de Mensagens de Tempo-Real

Para que ocorra o recebimento de uma mensagem de tempo real é necessário que a primitiva `Kernel_MessageRTReceive()` seja chamada, passando como parâmetro o ponteiro para a instância de `MessageRT`, assim como um ponteiro apontando para uma região de memória destinada à mensagem recebida e um outro apontando a uma região destinada ao identificador GlobalAccessPoint do processo emissor. Tal implementação pode ser observada na Figura 41.

```
void
Task_2_Implementation()
{
      char** ReceivedMessage = (char**)
            Kernel_MemoryAllocate(sizeof(char*));
      RTMessage_T* MessageInstance = (RTMessage_T*)
            Kernel_MemoryAllocate(sizeof(RTMessage_T));
      GlobalAccessPoint_T* GlobalAccessPoint = (GlobalAccessPoint_T*)
            Kernel_MemoryAllocate(sizeof(GlobalAccessPoint_T));

      Kernel_MessageRTInit(MessageInstance,2);
      Kernel_MessageRTReceive(MessageInstance,
                              ReceivedMessage,
                              GlobalAccessPoint);
      printf("\nFrom: %d, Msg: %s",
             *GlobalAccessPoint,*ReceivedMessage);
      fflush(stdout);

      Kernel_MessageRTFinish(MessageInstance);
}
```

Figura 41: Exemplo de Recebimento de Mensagem de Tempo-Real

Caso o PriorityCeiling contido na mensagem for superior à prioridade dinâmica do processo, o processo herdará a prioridade da mensagem recebida. Caso contrário, o processo continuará com a mesma prioridade dinâmica.

Após o termino da comunicação, isto é, após a chamada ao método nativo `kernel_MessageRTFinish()`, o processo terá sua prioridade dinâmica atualizada com o valor máximo do priority ceiling de todas outras instâncias de RTMessage. Caso não haja outras instâncias, sua prioridade dinâmica será atualizada com o valor da prioridade estática.

7.12 Prefetch

Assim como descrito na seção 6.4.4, o kernel SaoCarlOS suporta a técnica de Prefetching para garantir que o cadastro dos Identificadores dos Pontos de Acesso ocorram durante a inicialização do sistema.

A chamada a esta primitiva deve ocorrer após a chamada `Kernel_KernelInit()` e antes da chamada `Kernel_KernelStart()`. O Valor de retorno desta primitiva é KERNEL_SUCCESS, quando a comunicação com o Servidor de Nomes foi efetiva e foi possível recuperar os dados referentes ao Ponto de Acesso passado como parâmetro. O valor de retorno é `KERNEL_FAIL` caso a comunicação com o Servidor falhe ou quando os dados referentes ao ponto de acesso não possam ser recuperados do servidor. Um exemplo da implementação da técnica de prefetch no SaoCarlOS pode ser observada na Figura 42.

```
int
main()
{
   Process_T *Process_1 = (Process_T *)
      Kernel_MemoryAllocate(sizeof(Process_T));

   Kernel_KernelInit();

   Kernel_MessageRTPrefetch(2);

   Process_1->Priority = 15;

   Kernel_ProcessCreate(Process_1, Task_1_Implementation);

   Kernel_KernelStart();
}
```

Figura 42: Exemplo da Técnica de Prefetch

7.12 Memória Compartilhada

Assim como mencionado anteriormente, o escopo das variáveis utilizadas no SaoCarlOS é limitado a cada Processo. Deste modo, para que exista uma região de memória comum entre dois processos é necessária a declaração de uma memória compartilhada. Um exemplo da criação de memória compartilhada pode ser observado na Figura 43.

Para isto, é necessária a declaração de uma variável global do tipo `void*` e de uma instância local do tipo `SharedMemory_T*`, alocar memória e inicializar os valores de `SharedMemoryKey` e `SharedMemorySize` da instância. Após isto, basta

chamar o método `Kernel_MemorySharedAllocate()`, direcionando a saída desta função ao ponteiro para void. O retorno de tal função é um ponteiro para a memória compartilhada em questão. Esta chamada mapeia a funcionalidade de shmget() e shmat() do Linux.

```
void* SharedMemory;

int
main()
{
   SharedMemory_T* SharedInstance;
   Kernel_KernelInit();

   /* Initialize Kernel and User Process Communication Channel */
   SharedInstance = (SharedMemory_T*)
      Kernel_MemoryAllocate(sizeof(SharedMemory_T));
   SharedMemory = Kernel_MemorySharedAllocate(KernelSharedInstance,
                                              80);
   Kernel_KernelStart();
}
```

Figura 43: Alocando Memória Compartilhada

7.13 Recurso

Uma outra maneira de garantir a exclusão mútua no SaoCarlOS, é através da utilização de um Recurso. Tal recurso garante a dois processos acessar um dispositivo compartilhado através de Bloqueio (Lock) e Desbloqueio (Unlock).

De modo a utilizar tal recurso, é necessária a alocação de uma instância global. Após isto, deve-se passar esta instância para a função `Kernel_ResourceCreate()`. A partir de então, o recurso está pronto para ser utilizado através das chamadas `Kernel_ResourceLock()` e `Kernel_ResourceUnock()` que garantem o acesso à região crítica entre estas chamadas a apenas uma tarefa por vez. As chamadas de bloqueio e desbloqueio do Recurso no SaoCarlOS foram feitas empregando as primitivas Up e Down de um Semáforo.

Depois de terminada a utilização de um recurso, pode-se liberar as estruturas por ele utilizadas através da chamada. Uma exemplificação do recurso é demonstrada na Figura 44.

```
#include "SaoCarlOS.h"

Resource_T *ResourceInstance;

void
Task_1_Implementation()
{
   Kernel_ResourceLock(ResourceInstance);
   /* Critical Region */
   Kernel_ResourceUnlock(ResourceInstance);
}

int
main()
{
   Process_T *Process_1 = Kernel_MemoryAllocate(sizeof(Process_T));
   ResourceInstance = Kernel_MemoryAllocate(sizeof(Resource_T));

   Kernel_KernelInit();
   Process_1->Priority = 1;
   Kernel_ProcessCreate(Process_1, Task_1_Implementation);

   Kernel_ResourceCreate(ResourceInstance);

   Kernel_KernelStart();
}
```

Figura 44: Utilização de Bloqueio e Desbloqueio de um Recurso

7.14 Caixa de Mensagens (Mailbox)

Uma outra maneira de comunicação entre processos no SaoCarlOS é através de uma Caixa de Mensagem (ou *Mailbox*). Esta caixa de mensagens possui número e tamanho de blocos definidos pelo programador da Aplicação. Uma particularidade desta técnica é que tanto a leitura de uma caixa de mensagens vazia quanto a escrita em uma caixa de mensagens cheia irá causar o bloqueio do processo que tentou ler/escrever no *Mailbox*.

A utilização da caixa de mensagens pode ser feita de uma forma simples. Para criação do *mailbox* é necessário definir respectivamente o tamanho e o número

dos blocos através da chamada de sistema `Kernel_MailboxCreate()`. Note que para esta chamada é necessária a passagem da instância global já alocada.

```
#include "SaoCarlOS.h"
Mailbox_T *MailboxInstance;

void
Task_1_Implementation()
{
   char* ReadBuffer = (char*) Kernel_MemoryAllocate(2);
   Kernel_MailboxRead(MailboxInstance,ReadBuffer);
   ReadBuffer[1] = 0;
   printf("\nRead(%s)",ReadBuffer);
}

void
Task_2_Implementation()
{
   Kernel_MailboxWrite(MailboxInstance,"0");
}

int
main()
{
   Process_T *Process_1 = Kernel_MemoryAllocate(sizeof(Process_T));
   Process_T *Process_2 = Kernel_MemoryAllocate(sizeof(Process_T));
   MailboxInstance = Kernel_MemoryAllocate(sizeof(Mailbox_T));

   Kernel_KernelInit();
   Process_1->Priority = 1;
   Process_2->Priority = 1;
   Kernel_ProcessCreate(Process_1, Task_1_Implementation);
   Kernel_ProcessCreate(Process_2, Task_2_Implementation);

   Kernel_MailboxCreate(MailboxInstance,1,3); //Size,TotalBlocks

   Kernel_KernelStart();
}
```

Figura 45: Utilização de Troca de Mensagens utilizando uma Caixa de Mensagens

Após sua inicialização, a troca de mensagens entre processos de uma mesma estação pode ocorrer através das primitivas `Kernel_MailboxWrite()` e `Kernel_MailboxRead()`. Para ambas é necessária a utilização de um buffer de leitura/escrita do mesmo tamanho do bloco definido pela primitiva `Kernel_MailboxCreate()`.

Após a finalização da utilização da Caixa de Mensagens é possível liberar a memória por ele utilizada através da função `Kernel_MailboxTerminate()`. Um exemplo da utilização de um Mailbox no SaoCarlOS pode ser observado na Figura 45.

7.13 Conclusões do Capítulo

Neste capítulo pudemos observar as interfaces de programação do SaoCarlOS. Também foi possível entender melhor o funcionamento das primitivas de troca de mensagens, assim como o efeito das primitivas MessageRT sobre a execução de uma determinada tarefa.

Tais informações serão de grande valia para o capítulo posterior, pois nele serão avaliadas a troca de mensagens utilizando as técnicas de nomes globais, servidor de nomes, cache e prefetching.

Capítulo 8
Testes e Avaliação

De modo a medir e comparar o desempenho do sistema implementado foram desenvolvidos programas que exercitam as funcionalidades descritas nos capítulos anteriores. Tal validação proporcionará comparar o desempenho da utilização do Centralizador de Processos Globais e da utilização de Nomes Globais; o desempenho da Tabela Local de Identificadores Globais e da técnica de pre-fetching durante a inicialização do sistema. Nas próximas subseções estão descritos os procedimentos e resultados estatísticos obtidos através da ferramenta MinitTAB durante a fase de testes do SaoCarlOS.

8.1 Nomes Globais

8.1.1 Procedimento Utilizado

Para validar a implementação de Nomes Globais no Kernel SaoCarlOS as interfaces Kernel_NetworkUDP foram utilizadas. A partir delas foram gerados dois programas que exercitam a troca de mensagens 2.000 vezes. A implementação de ambas as tarefas podem ser observadas nas Figuras 46 e 47.

```
void
Task_1_Implementation()
{
   NetworkUDP_T* Instance = (NetworkUDP_T*)
      Kernel_MemoryAllocate(sizeof(NetworkUDP_T));
   char* Buffer = (char*)
      Kernel_MemoryAllocate(NETWORK_MSG_BUFFER_SIZE);
   int i = 1000;

   Kernel_NetworkUDPCreate(Instance,4002);
   Kernel_NetworkUDPSetRemote(Instance,"node1-data",4001);
   while(i--)
   {
      Kernel_NetworkUDPReceive(Instance,Buffer);
      Kernel_NetworkUDPSend(Instance,"Hello World.");
   }
   Kernel_NetworkUDPTerminate(Instance);
}
```

Figura 46: Validação de Nomes Globais Tarefa 1

```
void
Task_2_Implementation()
{
   NetworkUDP_T* Instance = (NetworkUDP_T *)
      Kernel_MemoryAllocate(sizeof(NetworkUDP_T));
   char* Buffer = (char*)
      Kernel_MemoryAllocate(NETWORK_MSG_BUFFER_SIZE);
   KernelTime_T StartTime = Kernel_TimeGetTime();
   int i = 1000;

   Kernel_NetworkUDPCreate(Instance,4001);
   Kernel_NetworkUDPSetRemote(Instance,"node2-data",4002);
   while(i--)
   {
      Kernel_NetworkUDPSend(Instance,"Hello World.");
      Kernel_NetworkUDPReceive(Instance,Buffer);
   }
   Kernel_NetworkUDPTerminate(Instance);
   printf("\nExecution Time: %d",
      Kernel_TimeGetTime()-StartTime);
}
```

Figura 47: Validação de Nomes Globais Tarefa 2

8.1.2 Resultados Obtidos

Os tempos de execução para 1000 envios e recebimentos de mensagens podem ser encontrados no Anexo B deste trabalho. A partir destes valores, podemos observar a dispersão dos tempos medidos em milisegundos pela Figura 48.

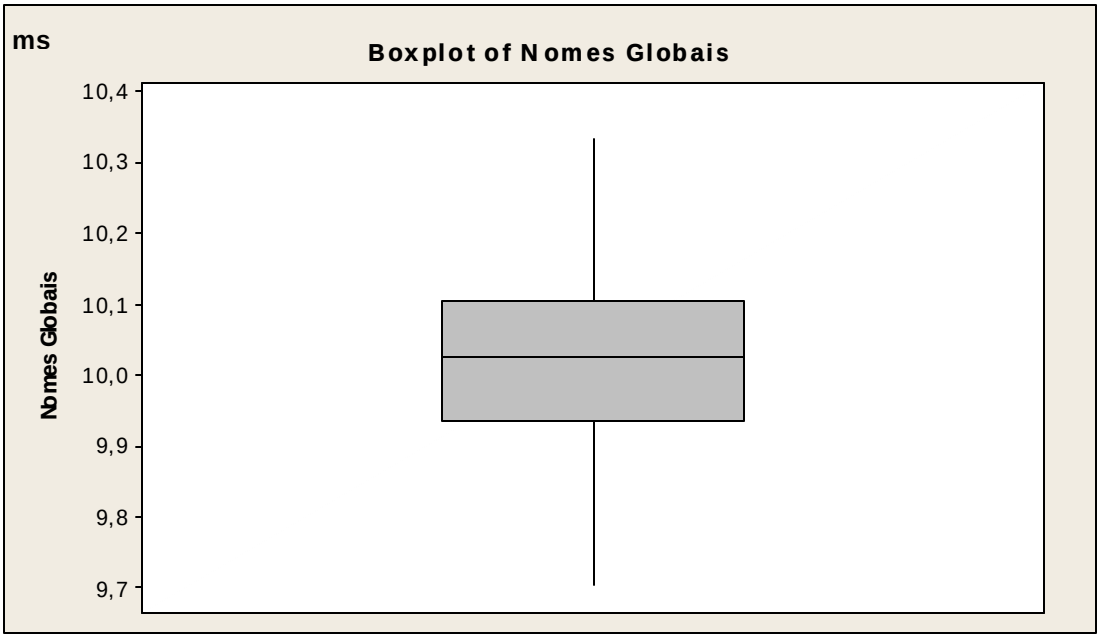

Figura 48: Boxplot da Distribuição de Tempos de transmissão de Mensagens com o uso de Nomes Globais

Como os valores foram obtidos através de seqüências de médias de 2000 trocas de mensagens, podemos também avaliar o histograma da distribuição, comparando-o a uma curva normal. Tal comparação pode ser observada pela Figura 49.

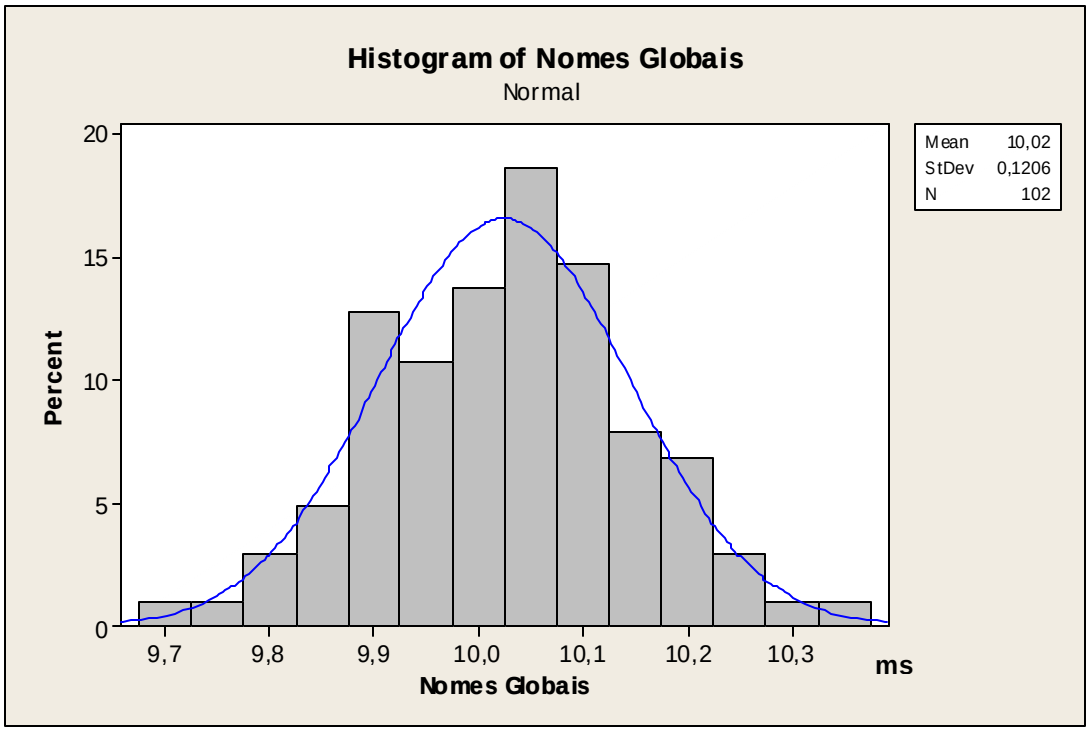

Figura 49: Histograma dos Tempos de Transmissão de Mensagens com o uso de Nomes Globais

No canto superior direito da Figura 49 podemos observar que o tempo de transmissão de uma mensagem leva em média um valor equivalente a 10,02 milisegundos variando de acordo com um e o Desvio Padrão de 120,6 microsegundos.

Também observamos a partir da Figura 49 que o histograma dos tempos de transmissão utilizando a técnica de nomes globais se aproxima de uma curva normal. Porém, antes de prosseguir com a analise, devemos garantir que estes dados se aproximam de uma curva normal. Deste modo, podemos fazer um teste de normalidade destes valores, como observado na Figura 50.

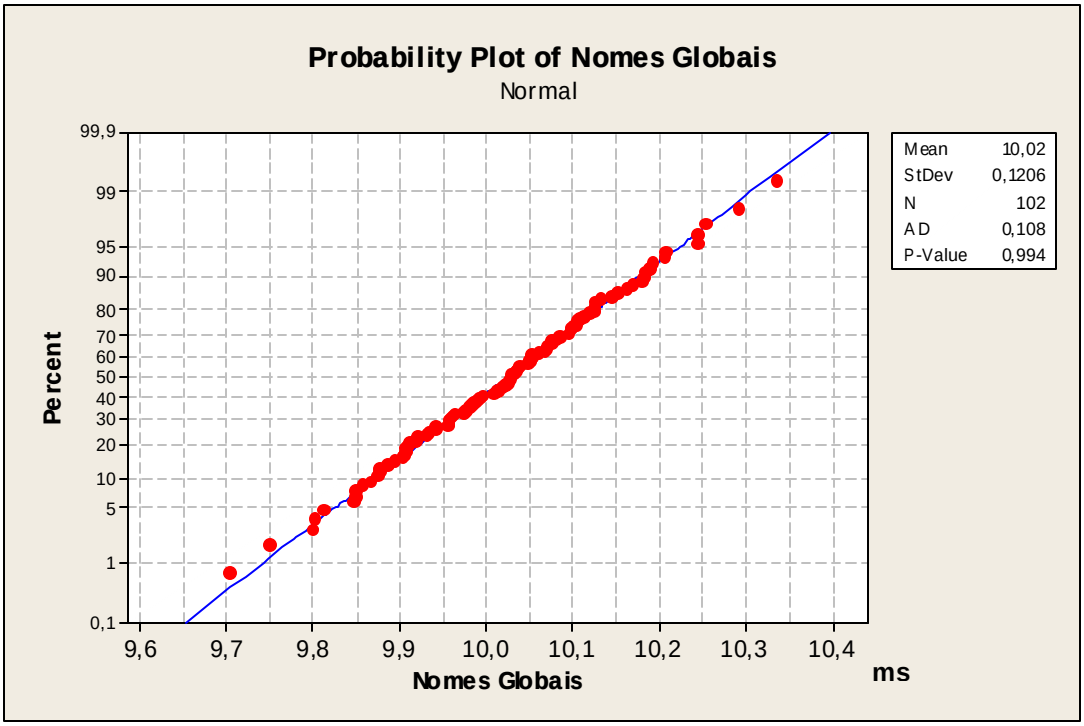

Figura 50: Teste de Normalidade para Tempos de Transmissão de Mensagem com o uso de Nomes Globais

Podemos verificar pelo visual da distribuição das ocorrências de tempos que os valores se aproximam da reta que caracteriza a distribuição como normal. Também é possível analisar a normalidade de uma distribuição através do Pvalue, salientando que valores superiores a 0,05 caracterizam uma distribuição como normal.

Neste caso específico, observamos o Pvalue com 0,994. Portanto, a distribuição das médias dos tempos de transmissão de mensagens utilizando a técnica de Nomes Globais pode ser dita uma distribuição Normal.

Um outro teste que podemos fazer é analisar seqüencialmente os dados dentro de um limitante superior de +3 desvios padrão (σ) e um limitante inferior de –3 desvios padrão (σ). Este teste pode ser observado na Figura 51.

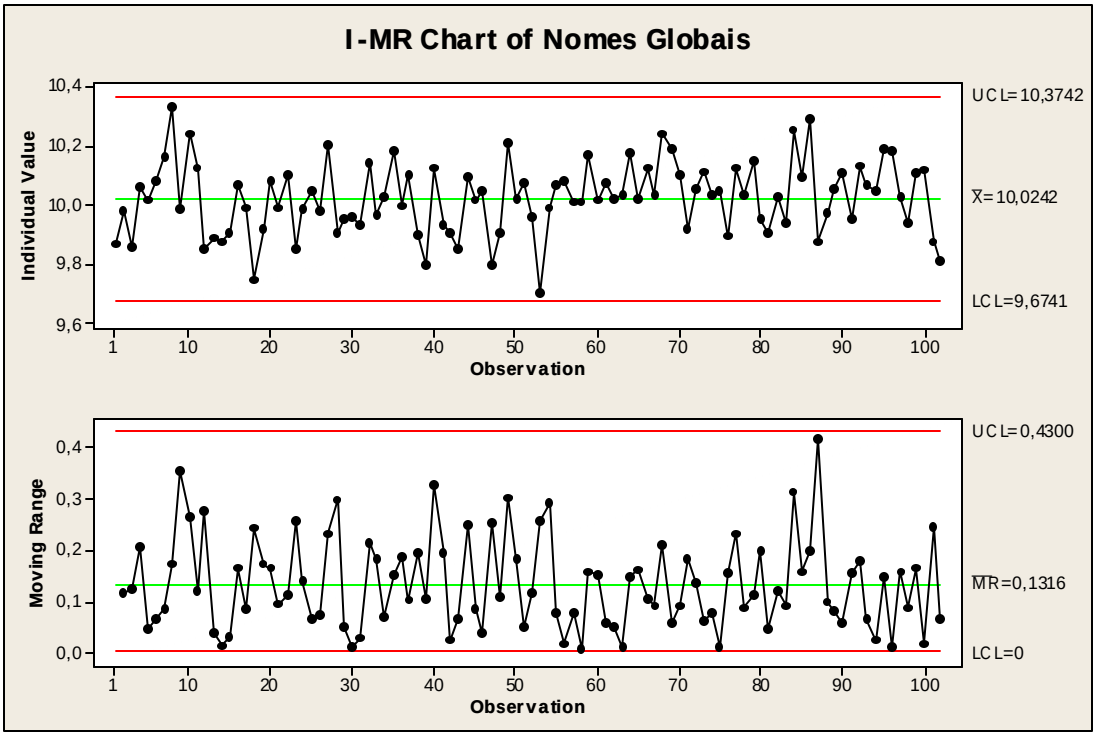

Figura 51: Analise dos Dados dentro de um Limite de +/- 3s

O primeiro gráfico mostra a variação dos dados de acordo com os valores de tempos coletados através do programa descrito em 8.1.1. Neste gráfico foi traçada a Média de X e os limitantes superior e inferior de +/- 3σ. Como podemos observar, nenhum dos pontos ultrapassou algum destes limites, determinando que o processo está sob controle. Segundo a estatística, tendo um processo sob controle, podemos garantir que a amostra não foi tendenciosa, e que é possível conseguir resultados semelhantes sempre que o mesmo processo for reproduzido.

No segundo gráfico observamos o intervalo de movimentação, calculado pela diferença entre valores tempos consecutivos. Também neste gráfico observamos a média do intervalo de movimentação um limite superior de +3σ, e um limitante inferior de 0, já que não podemos atingir uma intervalo de movimentação negativo.

Sendo assim, observamos que no gráfico de intervalo de movimentação também não houve valores fora dos limites. Portanto, podemos concluir que a troca de mensagens utilizando a técnica de Nomes Globais é um processo sob controle.

8.2 Servidor de Nomes

8.2.1 Procedimento Utilizado

De modo a validar somente a implementação de troca de mensagens utilizando um servidor de nomes, utilizamos o kernel SaoCarlOS compilado com as seguintes modificações no arquivo settings.h:

```
#define KERNEL_CACHE_AVAILABLE           0
#define KERNEL_CACHE_CLEANUP_AVAILABLE   0
```

Estas modificações permitem que o servidor de nomes seja acessado toda vez que uma nova mensagem for transmitida.

Juntamente com estas modificações, foram executadas 3 instâncias do kernel SaoCarlOS contendo cada um deles as tarefas descritas nas Figuras 52, 53 e 54:

```
void
Task_1_Implementation()
{
   char** ReceivedMessage = (char**)
      Kernel_MemoryAllocate(sizeof(char*));
   RTMessage_T* MessageInstance = (RTMessage_T*)
      Kernel_MemoryAllocate(sizeof(RTMessage_T));
   GlobalAccessPoint_T* GlobalAccessPoint =
      (GlobalAccessPoint_T*)
      Kernel_MemoryAllocate(sizeof(GlobalAccessPoint_T));
   int i = 1000;

   Kernel_MessageRTInit(MessageInstance,1);
   while(i--)
   {
      Kernel_MessageRTReceive(MessageInstance,
         ReceivedMessage,GlobalAccessPoint);
      Kernel_MessageRTSend(MessageInstance,"Hello World",2);
   }
   Kernel_MessageRTFinish(MessageInstance);
}
```

Figura 52: Troca de Mensagens utilizando MessageRT Tarefa 1

```
void
Task_2_Implementation()
{
   char** ReceivedMessage = (char**)
      Kernel_MemoryAllocate(sizeof(char*));
   RTMessage_T* MessageInstance = (RTMessage_T*)
      Kernel_MemoryAllocate(sizeof(RTMessage_T));
   GlobalAccessPoint_T* GlobalAccessPoint =
      (GlobalAccessPoint_T*)
      Kernel_MemoryAllocate(sizeof(GlobalAccessPoint_T));
   KernelTime_T StartTime = Kernel_TimeGetTime();
   int i = 1000;

   Kernel_MessageRTInit(MessageInstance,1);
   while(i--)
   {
      Kernel_MessageRTReceive(MessageInstance,
         ReceivedMessage,GlobalAccessPoint);
      Kernel_MessageRTSend(MessageInstance,"Hello World",2);
   }
   Kernel_MessageRTFinish(MessageInstance);

   printf("\nExecution Time: %d",
      Kernel_TimeGetTime()-StartTime);
}
```

Figura 53: Troca de Mensagens utilizando MessageRT Tarefa 2

```
void
Task_3_Implementation()
{
   NamingServer_Init();
}
```

Figura 54: Tarefa Implementando Servidor de Nomes

8.2.2 Resultados Obtidos

Assim como na seção 8.1.2, os tempos de execução para 1000 execuções deste teste podem ser encontrados no Anexo B deste trabalho. A partir destes valores, podemos observar a dispersão dos dados pela Figura 55.

Note que a Figura 55 difere da Figura 49, pois a média da Distribuição da Figura 55 é por volta de 3 vezes superior à media da Distribuição da Figura 49. Este comportamento já era esperado, já que durante a transmissão de cada

mensagem, uma consulta ao servidor de nomes esta sendo realizada. Deste modo, existem 3 envios de mensagens:

- Mensagem de Requisição de Consulta gerada pelo transmissor da MensageRT e destinada ao Servidor de Nomes.
- Reconhecimento do Servidor de Nomes, contendo a resposta à requisição, gerada pelo Servidor de Nomes e destinada ao transmissor da MessageRT.
- Transmissão da MensageRT, gerada pelo transmissor da MessageRT e destinada ao receptor da MessageRT.

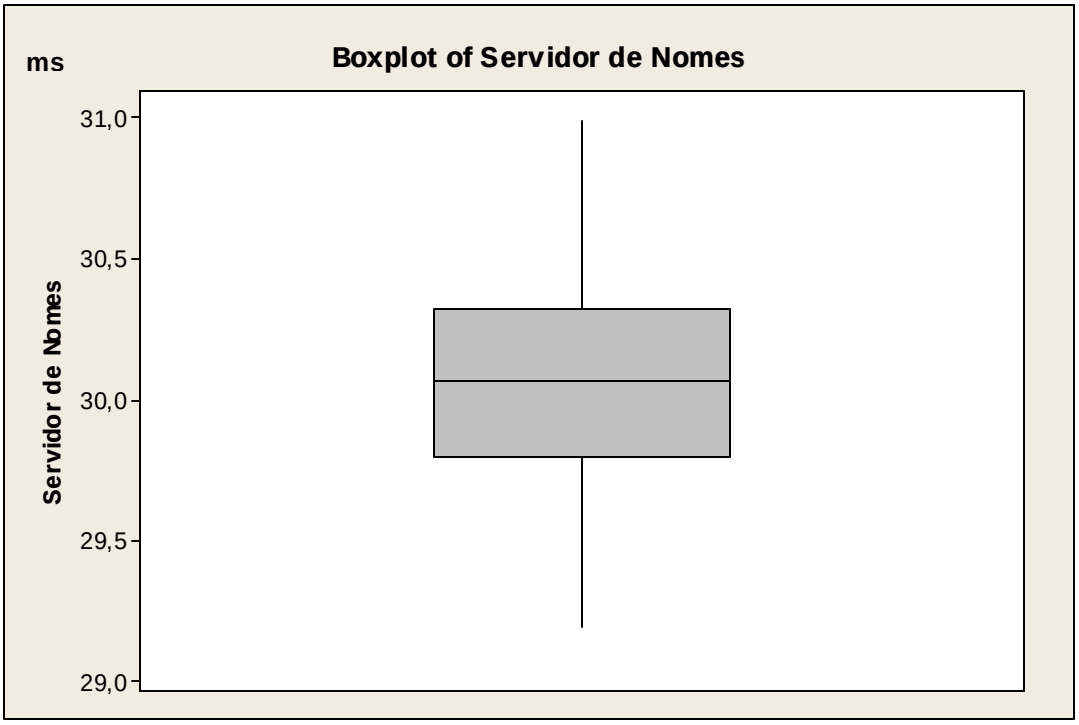

Figura 55: Boxplot da Distribuição de Tempos de Transmissão de MessageRT com o uso de Servidor de Nomes

Assim como feito na seção anterior, podemos também avaliar o histograma da distribuição, comparando-o a uma curva normal, já que estes valores foram obtidos através de seqüências de médias de 2000 trocas de mensagens. Tal comparação pode ser observada pela Figura 55.

Podemos observar na Figura 56 que o tempo de transmissão de uma MensageRT leva em média um valor equivalente a 30,07 milisegundos variando de acordo com um Desvio Padrão de 364 microsegundos.

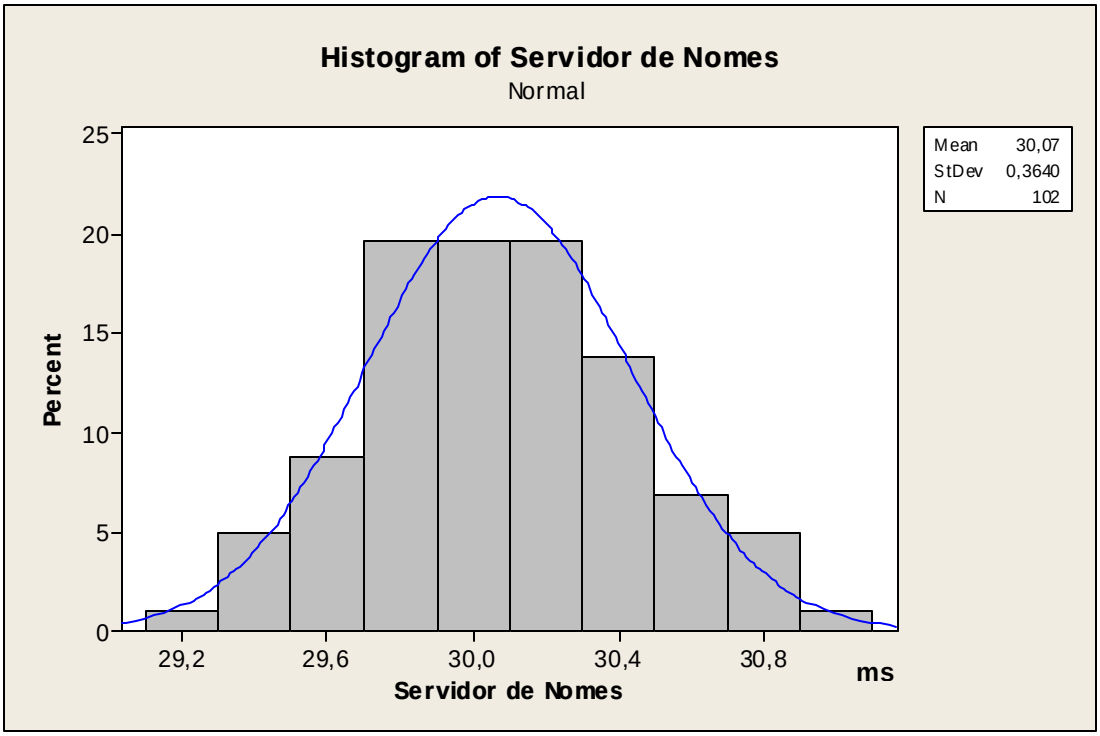

Figura 56: Histograma da Distribuição de Tempos de Transmissão de MessageRT com o uso de Servidor de Nomes

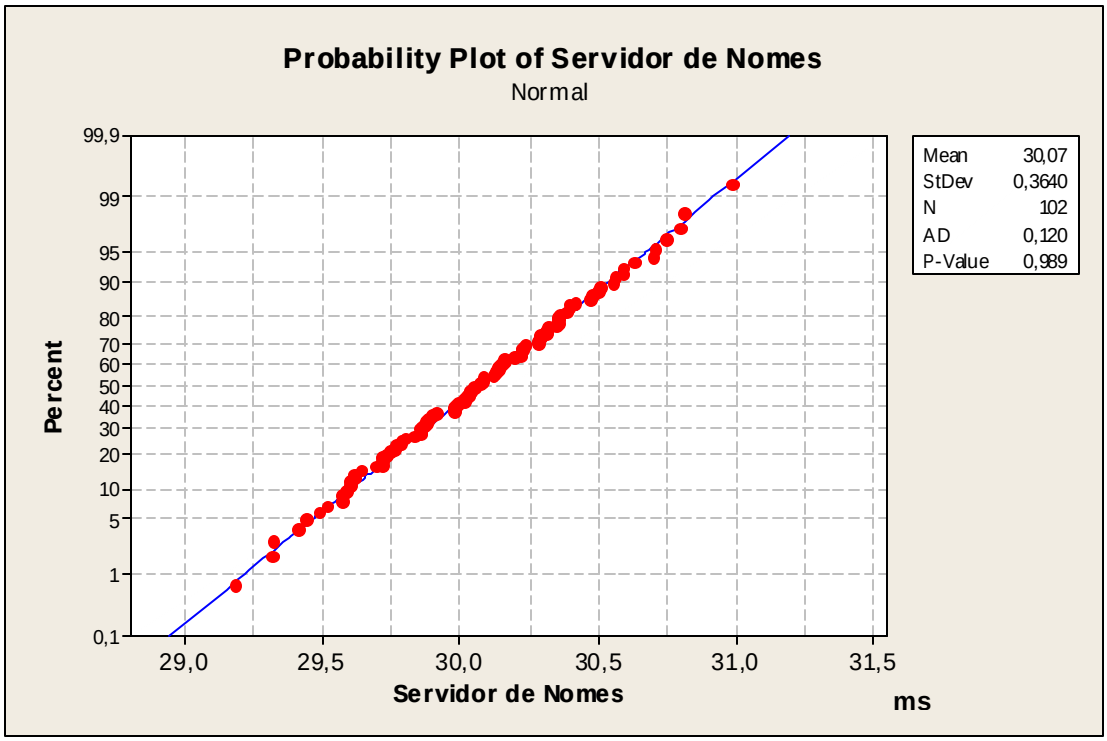

Figura 57: Teste de Normalidade para a Distribuição de Tempos de Transmissão de MessageRT com o uso de Servidor de Nomes

Assim como feito anteriormente, podemos aplicar o Teste de Normalidade sobre os valores apresentados nas Figuras 55 e 56. Tal teste de Normalidade pode ser observado na Figura 57 e a partir dele podemos identificar a distribuição dos tempos de transmissão de MessageRT utilizando apenas a técnica de Servidor de Nomes como sendo uma Normal, já que o Pvalue = 0,989 apresentado na Figura 57 é superior a 0,05.

Do mesmo modo como na seção anterior, podemos analisar na Figura 58 a seqüência dos dados dentro de um limitante superior de +3 desvios padrão (σ) e um limitante inferior de –3 desvios padrão (σ).

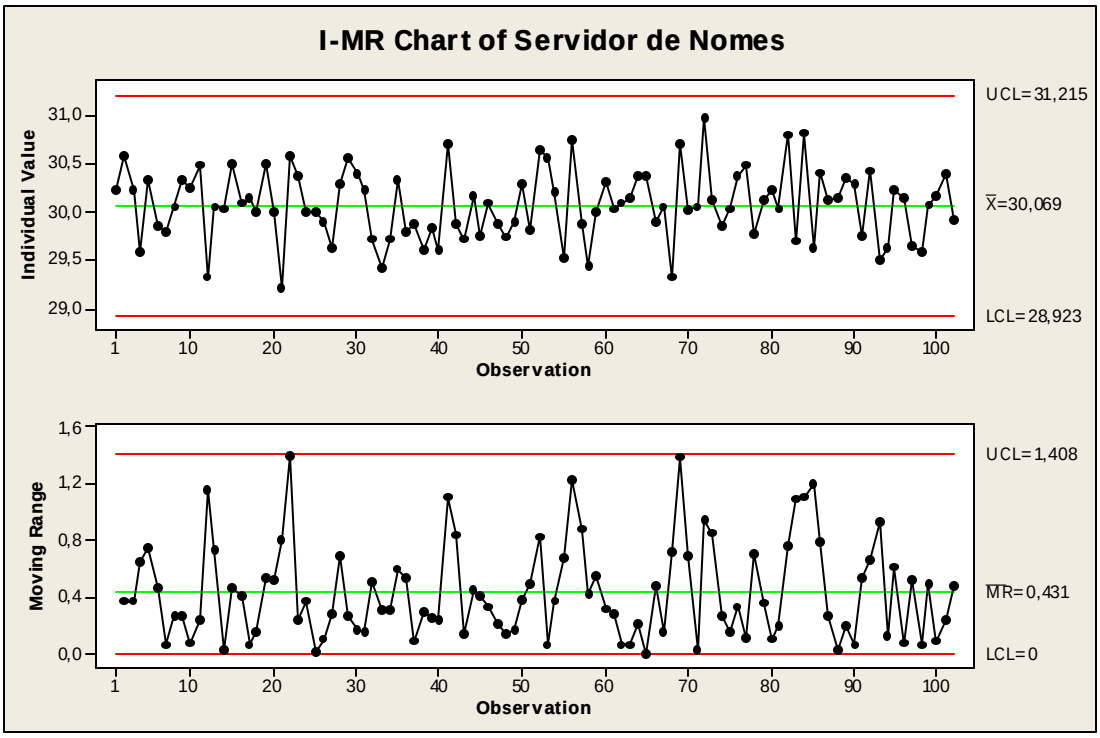

Figura 58: Análise dos Dados dentro de um Limite de +/- 3s

Assim como na seção 8.1.2, observamos que ambos os gráficos de valores individuais e intervalo de movimentação não apresentaram valores fora dos limites. Portanto, apesar de possuir um tempo médio superior ao tempo médio da técnica empregada na seção anterior, podemos concluir que a troca de mensagens utilizando a técnica de Servidor de Nomes é um processo sob controle.

8.3 Cache

8.3.1 Procedimento Utilizado

Assim como realizado na seção 8.2.1, de modo a validar somente a implementação de troca de mensagens utilizando um servidor de nomes e a técnica de caching das requisições ao servidor de nomes, utilizamos o kernel SaoCarlOS compilado com as seguintes modificações no arquivo settings.h:

```
#define KERNEL_CACHE_AVAILABLE           1
#define KERNEL_CACHE_CLEANUP_AVAILABLE   0
```

Estas modificações permitem que o servidor de nomes seja acessado apenas durante a primeira vez que uma mensagem for transmitida.

Também, além destas modificações, utilizaremos as mesmas 3 tarefas definidas nas Figuras 52, 53 e 54. Do mesmo modo que na seção 8.2.1, estas tarefas estarão distribuídas em 3 instâncias do kernel SaoCarlOS .

8.3.2 Resultados Obtidos

Diferentemente das duas seções anteriores, podemos observar na Figura 59 o Boxplot da distribuição de tempos de transmissão de MessageRTs apresenta um ponto fora da curva (ou Outlier). Este ponto fora da curva, cujo valor está próximo aos 30 milisegundos, é referente à transmissão da primeira mensagem, na qual ocorre a consulta ao Servidor de Nomes.

As demais transmissões não necessitam de consultas ao Servidor de Nomes, já que o cache local é suficientemente capaz de resolver o endereço do ponto de acesso global da tarefa destino. Sendo assim, os demais tempos de transmissão ficam aglutinados por volta de 10 milisegundos.

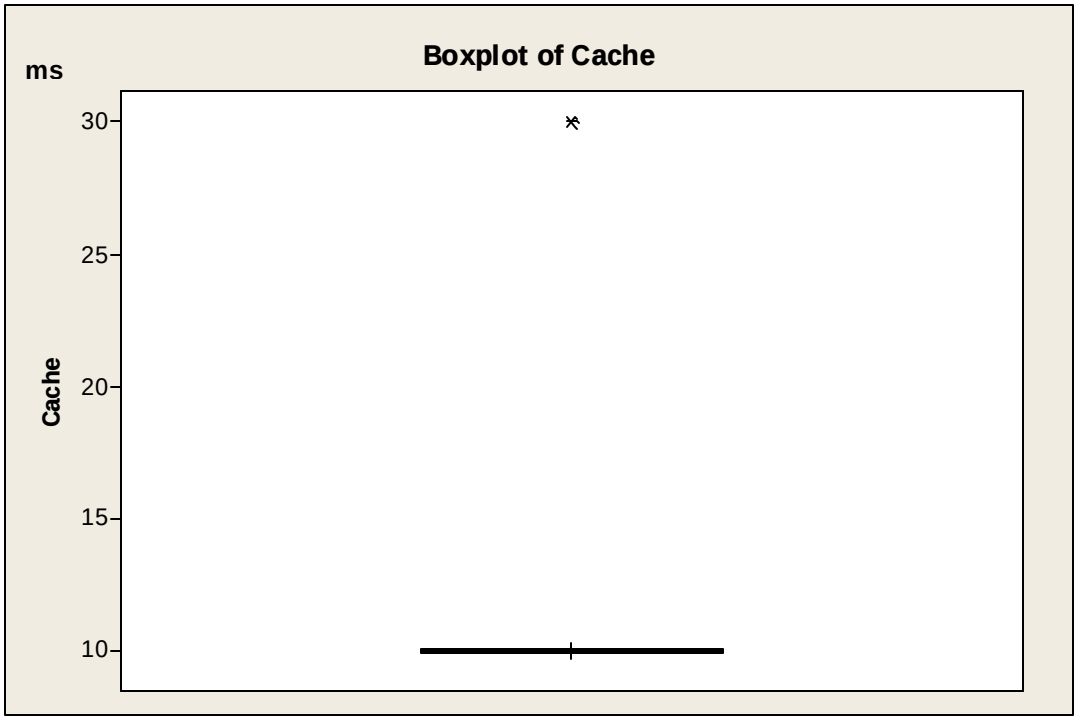

Figura 59: Boxplot da Distribuição de Tempos de Transmissão de MessageRT com o uso de Cache

Podemos também observar o Histograma da distribuição de tempos de transmissão de Mensagens de Tempo Real na Figura 60. Nesta figura observamos que o tempo de transmissão de uma MensageRT leva em média um valor equivalente a 10,22 milisegundos variando de acordo com um Desvio Padrão de 1,98 milisegundos.

Também, esta variação devido a um Outlier completamente fora da distribuição compromete o resultado do teste de normalidade para os tempos de transmissão de MessageRT. Podemos observar na Figura 61 que a reta traçada na tabela logarítmica não condiz com os pontos nela apresentados.

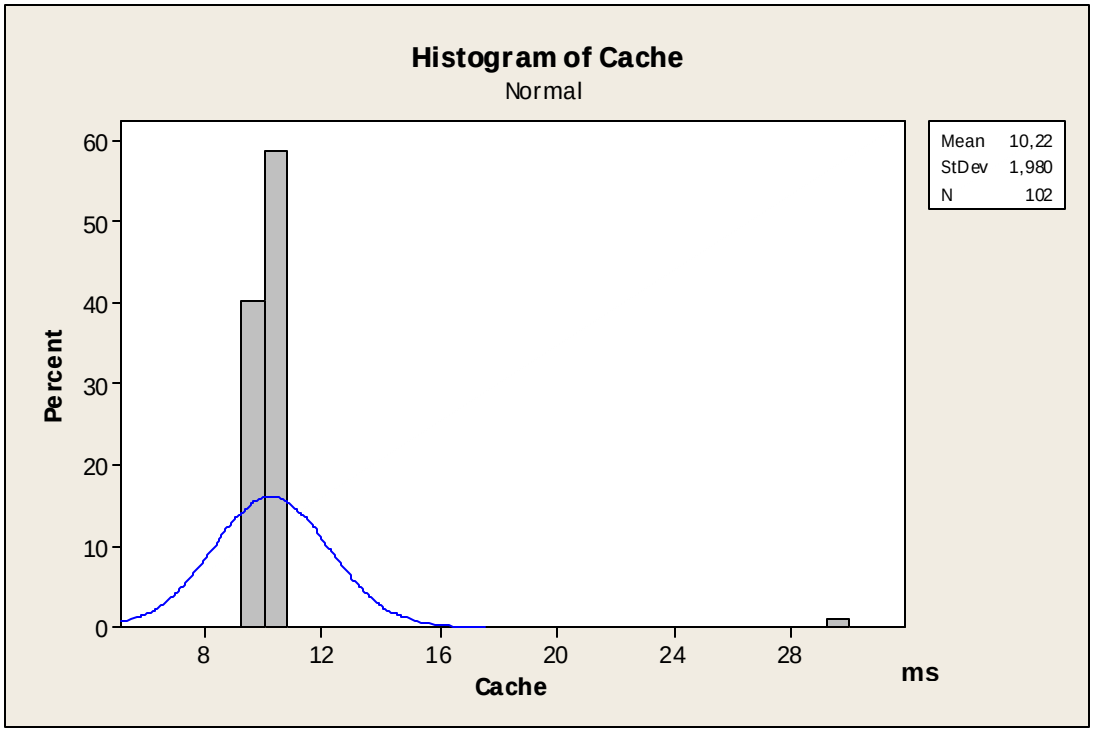

Figura 60: Histograma da Distribuição de Tempos de Transmissão de MessageRT com o uso de Cache

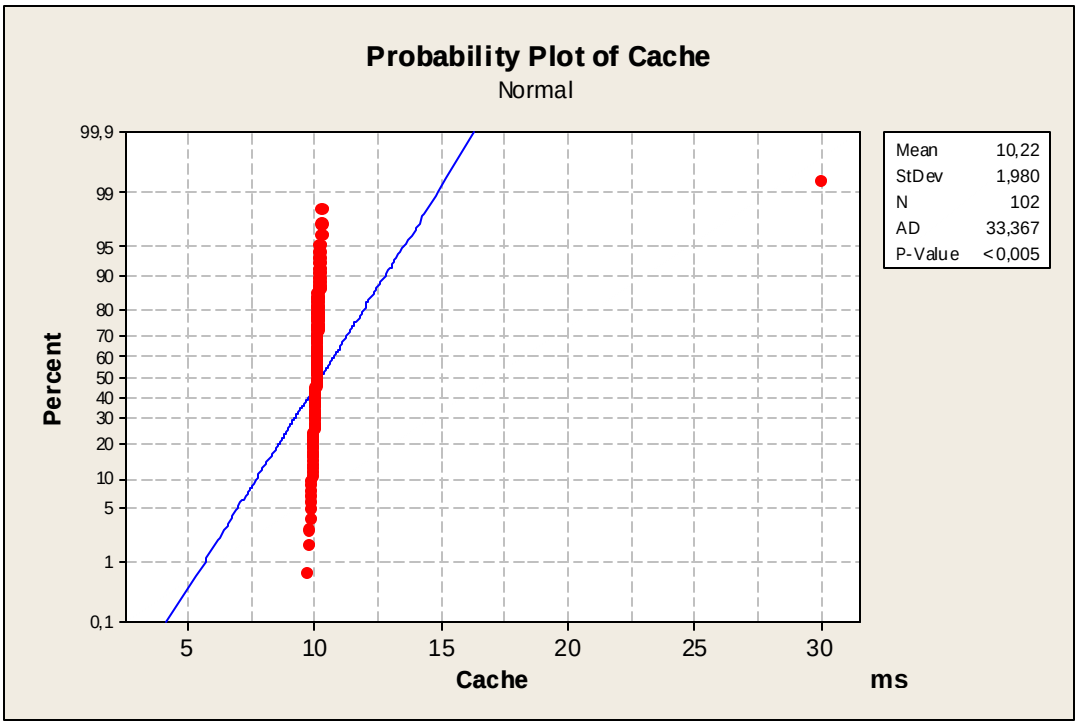

Figura 61: Teste de Normalidade para Distribuição de Tempos de Transmissão de MessageRT com o uso de Cache

Porém, executando o mesmo teste de normalidade, mas desta vez removendo o tempo da primeira transmissão, podemos observar pela Figura 62 que a distribuição dos demais tempos assume uma forma normal.

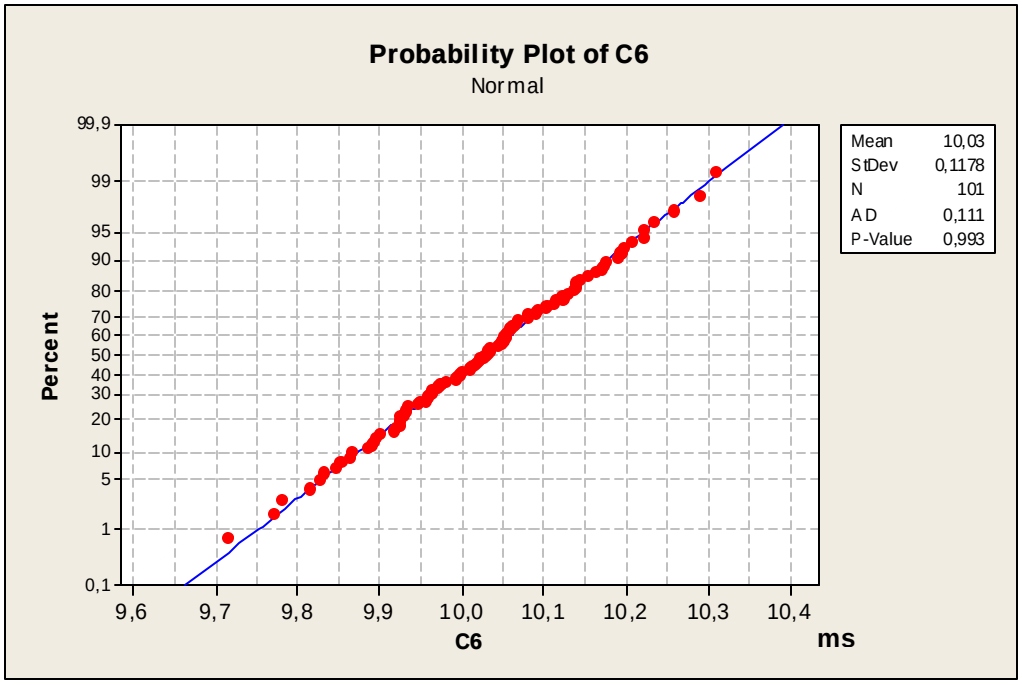

Figura 62: Teste de Normalidade da Distribuição dos Tempos de Transmissão de MessageRT removendo o Tempo de Acesso ao Servidor de Nomes

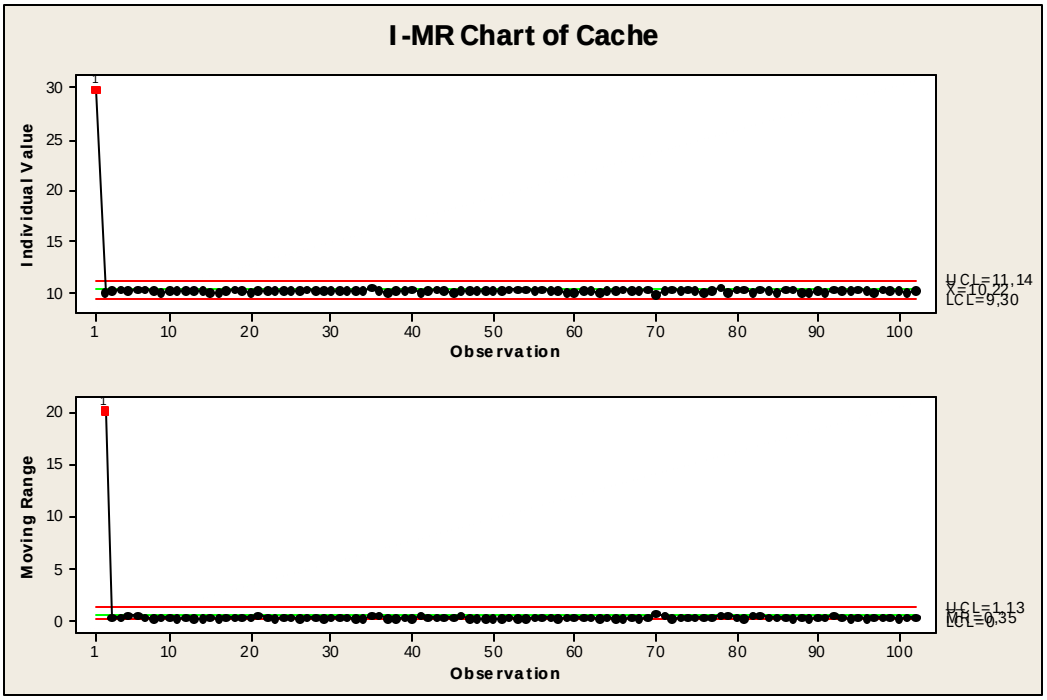

Figura 63: Análise dos Dados dentro de um Limite de +/- 3s

Podemos observar também na Figura 63 que, exceto para a primeira transmissão, todos os outros valores de tempo se mantêm dentro dos limites de +/- 3s. Porém, pela existência de um ponto fora dos limites de controle, podemos dizer que a transmissão de MessageRT utilizando um Servidor de Nomes e um Cache local é um processo fora de controle.

8.4 Prefetching

8.4.1 Procedimento Utilizado

Realizaremos a analise da técnica de Prefetching utilizando os mesmos procedimentos descritos na seção 8.3.1 e acrescentando apenas a chamada para a chamada de sistema Kernel_MessageRTPrefetching() logo após a chamada de Kernel_KernelInit(). Como parâmetro a função Kernel_MessageRTPrefetching estaremos passando o Ponto de Acesso Global da outra tarefa atuando como interlocutor na comunicação.

8.4.2 Resultados Obtidos

Podemos observar através do Boxplot descrito na Figura 64 que os tempos de transmissão de MessageRT utilizando Prefetching assumem valores muito próximos aos vistos anteriormente na Figura 48, onde representávamos a análise de troca de mensagens utilizando Nomes Globais.

Também, observando a Figura 65, contendo o Histograma dos tempos de transmissão utilizando Prefetching, podemos identificar uma semelhança com uma distribuição normal. Além disso, temos o tempo de transmissão de uma MensageRT levando em média um valor equivalente a 10,03 milisegundos variando de acordo com um e o Desvio Padrão de 121 microsegundos. Tais valores são muito próximos àqueles vistos durante a utilização de Nomes Globais (média de 10,02 milisegundos e Desvio Padrão de 120,6 microsegundos).

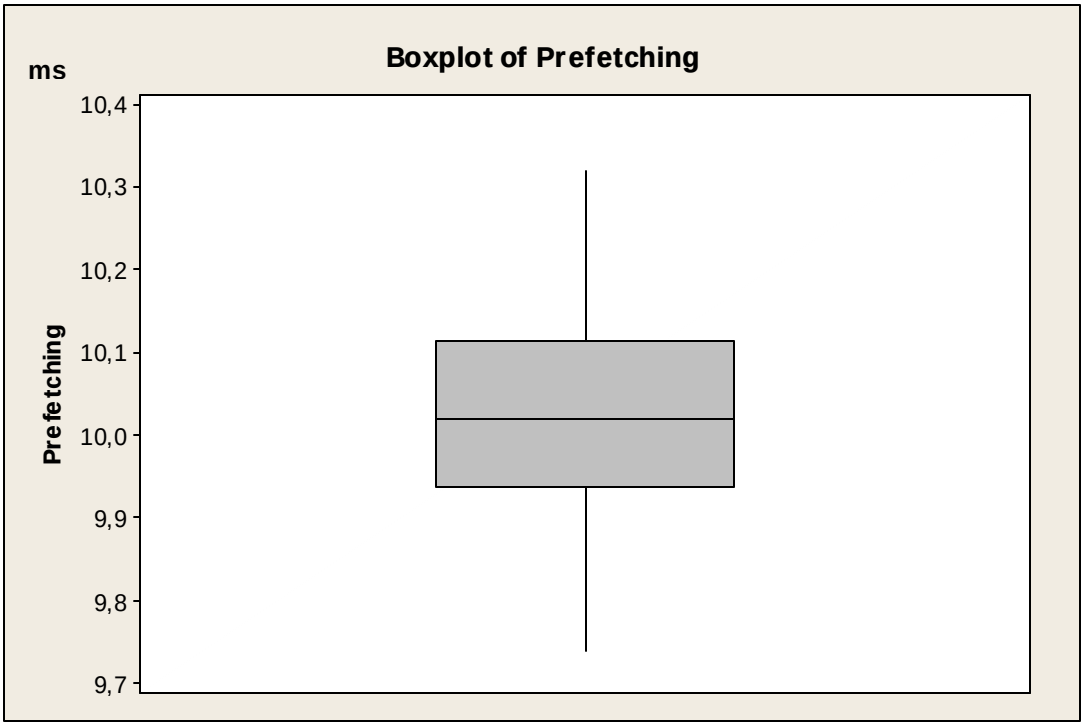

Figura 64: Boxplot da Distribuiçao dos Tempos de Transmissão de MessageRT com o uso de Prefetching

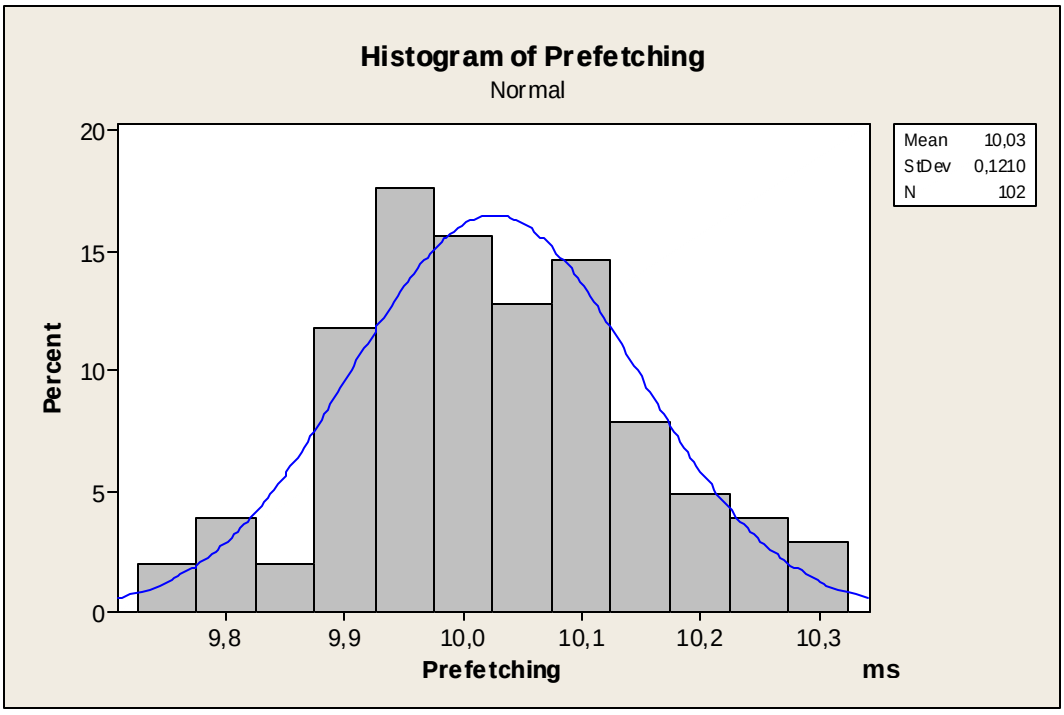

Figura 65: Histograma da Distribuição dos Tempos de Transmissão de MessageRT com o uso de Prefetching

Analisando ainda a Figura 66, contendo o teste de normalidade para os dados apresentados nesta seção, podemos verificar que, estando o Pvalue = 0,918 acima de 0,05, podemos classificar a distribuição dos tempos de transmissão de MessageRT como Normal.

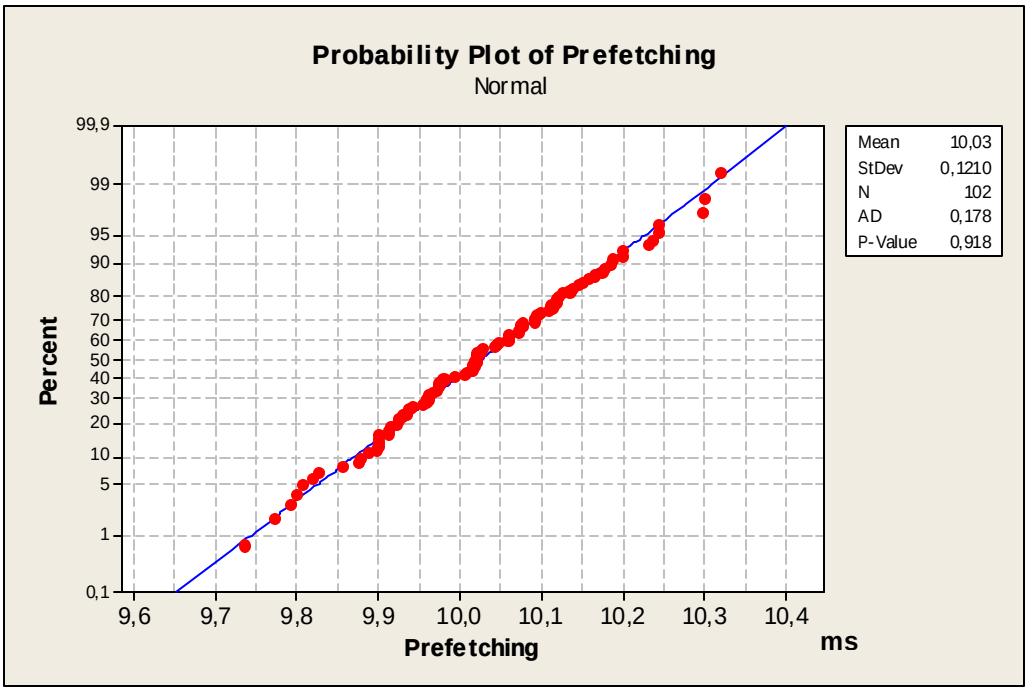

Figura 66: Teste de Normalidade para a Distribuição de Tempos de Transmissão de MessageRT com o uso de Prefetching

Também, analisando o Chart I-MR na Figura 67, podemos observar que o processo de transmissão de mensagens de Tempo Real utilizando a técnica de Prefetching é um processo sob controle, pois nenhum ponto ultrapassou nenhum limite de mais ou menos 3 desvios padrões durante a coleta de toda a amostra.

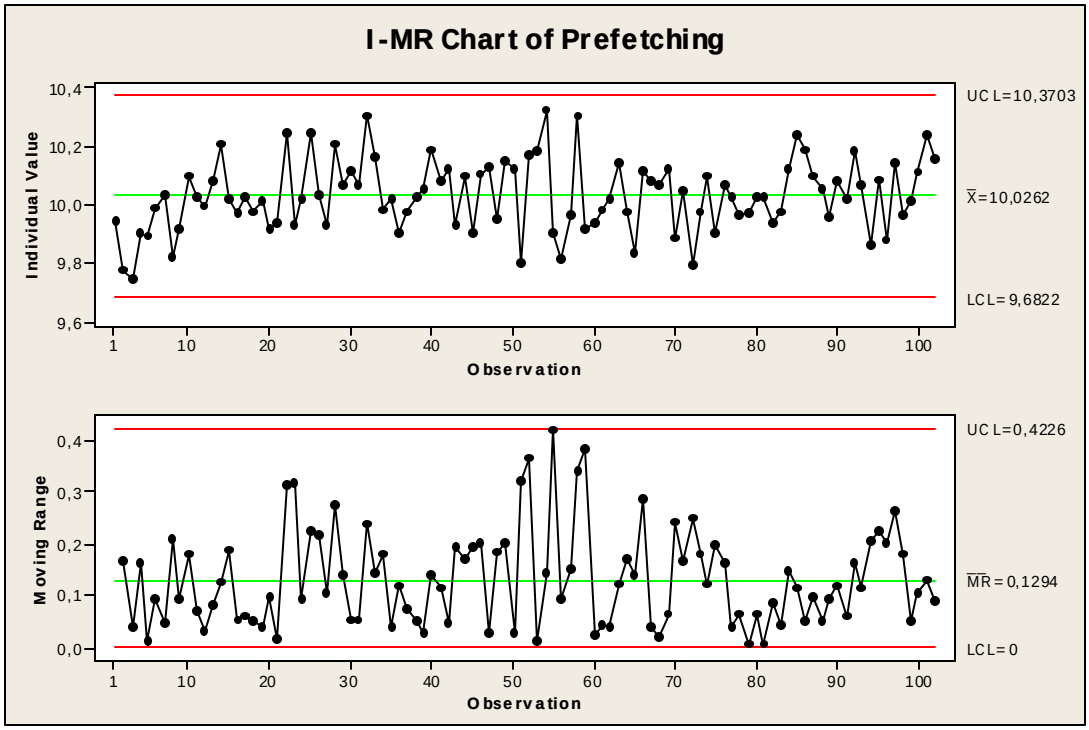

Figura 67: Análise dos Dados dentro de um Limite de +/- 3s

8.5 Conclusões do Capítulo

Neste capítulo pudemos observar e comparar as distribuições de tempo de transmissão utilizando 4 diferentes técnicas: Nomes Globais; Servidor de Nomes; Servidor de Nomes e Cache; Servidor de Nomes, Cache e Prefetching.

Pudemos também perceber que os tempos decorrentes da utilização de Servidor de Nomes, Cache e Prefetching se aproximam muito dos tempos encontrados utilizando Nomes Globais. Podemos melhor verificar este comportamento na Figura 66, com o Boxplot de todos os métodos alinhados.

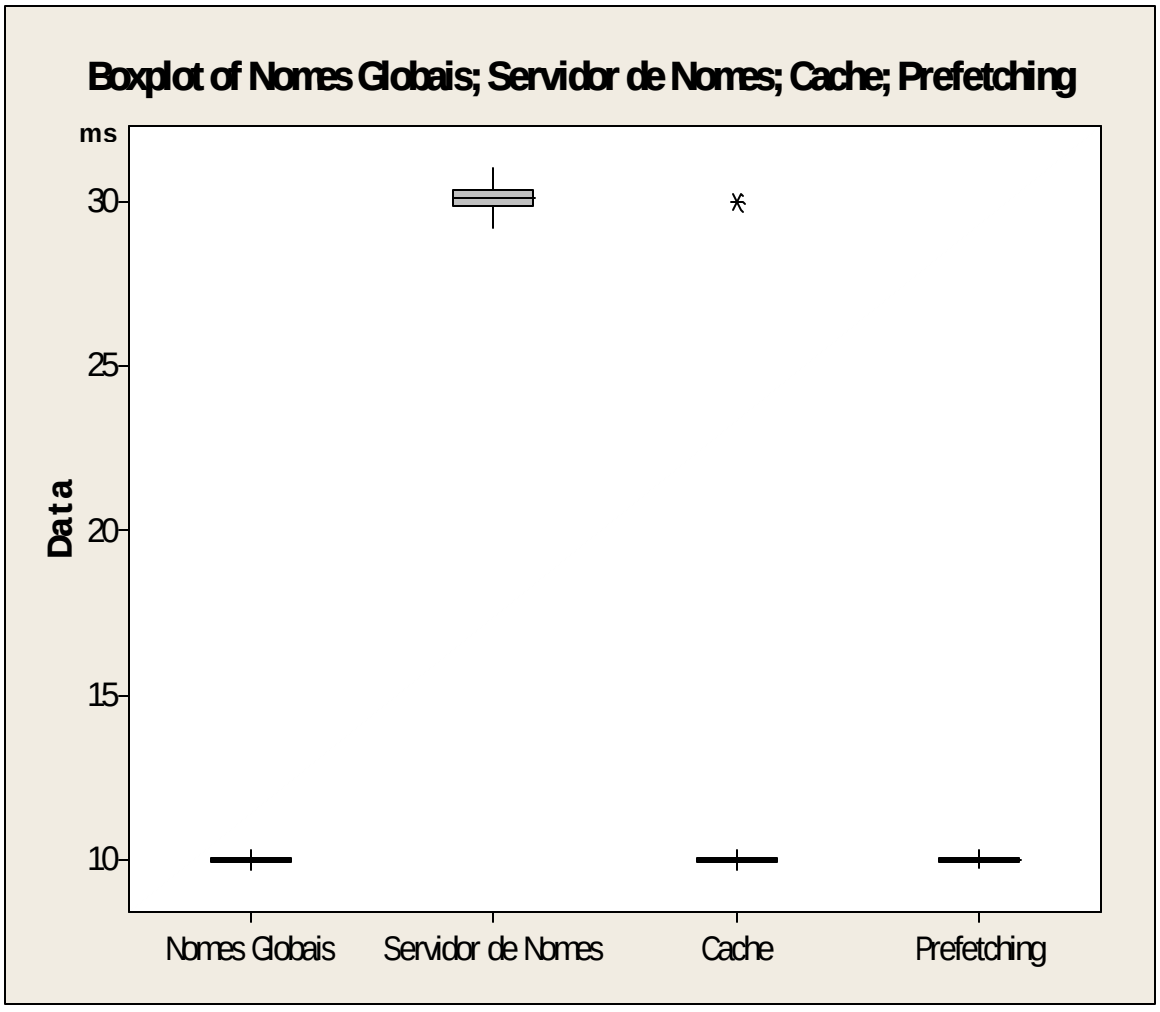

Figura 68: Boxplot das técnicas de Nomes Globais, Servidor de Nomes, Cache e Prefetching

Capítulo 9
Conclusões do Trabalho

Ao término deste projeto, obtivemos um Middleware e um sistema operacional, capaz de prover troca de mensagens utilizando-se de requisitos temporais. Devido à implementação do Priority Ceiling Protocol, podemos garantir que a troca de mensagens não causará inversão de prioridades ou deadlocks.

Também, a existência de um servidor de nomes provê maior transparência à localização de processos e à migração de processos no Sistema Operacional SaoCarlOS.

A Figura 69 compara o tempo e o número de mensagens de controle necessárias para o envio de 1024 bytes de um processo a outro em duas estações distintas. As técnicas utilizadas para coletar os dados são as mesmas técnicas descritas neste trabalho.

	Inicialização do Sistema	Primeira Troca Msg	Demais Trocas Msgs	Acesso a um Novo Serviço
Nomes Globais	0 msg / 0 ms	1 msg / 10 ms	1 msg / 10 ms	Não Disponível
Servidor de Nomes	0 msg / 0 ms	3 msg / 30 ms	3 msg / 30 ms	3 msg / 30 ms
Servidor de Nomes + Cache	0 msg / 0 ms	3 msg / 30 ms	1 msg / 10 ms	3 msg / 30 ms
Servidor de Nomes + Cache + Pre-Fetching	2 msg / 20 ms	1 msg / 10 ms	1 msg / 10 ms	3 msg / 30 ms

Figura 69: Mensagens Utilizadas para Uma Transmissão

Observando a Figura 69, é possível verificar que a utilização de um Servidor de Nomes, combinado a um Cache e a técnica de Pré-fetching, podemos alcançar um desempenho semelhante a um sistema utilizando nomes globais. Deste modo, podemos prover transparência e independência de localização sem causar muita sobrecarga ao sistema.

Também, os valores apresentados nesta figura são afetados por todas as sobrecargas de processamento apresentadas na seção 4.4 (Incertezas quanto ao

tempo de transmissão). Além da sobrecarga de transmissão sobre uma rede Giga Ethernet, os tempos apresentados na Figura 69 também são afetados pela sobrecarga decorrente do SaoCarlOS estar sendo executado e escalonado como um processo do Linux.

9.1 Trabalhos Futuros

Os próximos trabalhos envolvendo o SaoCarlOS estão contidos em cada um dos tópicos seguintes.

9.1.1 Programação Baixo Nível

Durante o desenvolvimento do SaoCarlOS, muitas funcionalidades já existentes no Linux foram reutilizadas, de modo a facilitar sua implementação. Um trabalho futuro seria promover a independência do SaoCarlOS de um sistema operacional hospedeiro.

9.1.2 Programação no Nível do Kernel

Poderá ainda ser criado um sistema de arquivos e uma interface de usuário (ou *shell*) para o SaoCarlOS.

9.1.3 Programação Alto Nível

Ainda é possível a criação de uma interface gráfica para desenvolvimento de aplicações de tempo-real, como o TEV (Teaching Environment for Virtuoso) desenvolvido no trabalho de [31].

Capítulo 10
Referências

[1]	HONG, S.; SEO, Y.; PARK, J. **ARX/ULTRA:** A New Real-Time Kernel Architecture for Supporting User-Level Threads. In: Technical Report SNU-EE-TR, 3, 1997, School of Electrical Engineering, Seoul National University, Seoul, Korea.
[2]	ABDELZAHER, T. F.; SHIN, K. G. **Optimal Combined Task and Message Scheduling in Distributed Real-time Systems**, In: Proceedings of the Sixteenth IEEE Real-time Systems Symposium 1995, Piza, Italy, 162-171.
[3]	ABDELZAHER, T. F.; SHIN, K. G. **Combined task and message scheduling in distributed real-time systems**. IEEE Transactions on Parallel and Distributed Systems, November 1999 10-11
[4]	KITAYAMA, Takuro; NAKAJIMA T.; TOKUDA, H. **RT-IPC:** An IPC Extension for Real-Time Mach. In Proceedings of the 2nd Microkernel and Other Kernel Architectures. USENIX, 1993.
[5]	DRUSCHEL, P.; BANGA, G.; **Lazy Receiver Processing (LRP):** A Network Subsystem Architecture for Server Systems, in Proceedings of the 1 st Symposium on Operating Systems Design and Implementation, USENIX Association, October 1996.
[6]	CAO, P.; LIU, C. **Maintaining Strong Cache Consistency in the World Wide Web** IEEE Transactions on Computers 47, Apr.1998, 445-457.
[7]	RAMANATHAN, P.; SHIN, K. G. **Delivery of Time-Critical Messages using a Multiple Copy Approach** - ACM Trans. on Comuter Systems Magazine, vol 10, no. 2, May 1992, pp. 144-166
[8]	HUTCHISON, D.; MERABTI, M. **Ethernet for Real-Time Applications**, IEE Proceedings 134 47-53 Jan. 1987.
[9]	LANN, G. **The 802.3D Protocol:** A Variation on the IEEE 802.3 Standard for Real-Time LANs, INRIABP 105, 78153, Le Chesnay Cedex, France (1987).

[10]	TOKUDA, H et al **Priority Inversions in Real-Time Communication**, Proc. of the 10th IEEE Real-Time Systems Sym-posium, pp. 348-359 (1989).
[11]	TANENBAUM, A. S. **Modern Operating System** - Prentice Hall 1992
[12]	MACHADO, F. B.; MAIA, Luiz P. **Arquitetura de Sistemas Operacionais** – Editora LTC –– 1997
[13]	LEVI, S.; AGRAWALA, A. K. **Real-Time System Design** - McGraw-Hill - 1990
[14]	TANENBAUM, A. S. **Computer Networks** – Prentice Hall – 1996
[15]	COMER, D. E. **Internetworking with TCP/IP vol. 1** - Prentice Hall– 1995
[16]	STEVENS, W. R **TCP/IP Illustrated vol. 1** Addison-Wesley - 1997
[17]	DIJKSTRA, E. W. **Co-operating Sequential Processes** in Programing Languages. Genuys, F (Ed), London: Academic Press, 1965
[18]	COFFMAN, E.G.; ELPHICK, M.J.; SHOSHNI, A. **System Deadlocks** Computer Surveys, vol 3, páginas 67 e 68, Junho de 1971
[19]	LEUNG, J.; WHITEHEAD, J. **On the complexity of Fixed Priority Scheduling of periodic, real time tasks**, Performance Evaluation (Netherlands) 2(4), 237-50 (1982)
[20]	LIU, C.; LAYLAND, J. **Scheduling algorithms for multiprograming in hard real-time enviroment** JACM, 20(1), 46-61 (1973)
[21]	FLYNN, M. **Some Computer Organizations and Their Effectiviness**, IEEE Trans. On Computers, vol C-21, pp. 948-960, Setembro 1972
[22]	BRADEN, R. **Requirements for Internet Hosts:** Communication Layers, STD 3, RFC 1122, Outubro 1989
[23]	CLARK, D. **Window and Acknowledgment Strategy in TCP**, RFC 813, Julho 1982

[24]	IEEE Standard 1003.4 – Real Time Extension
[25]	LAUER H.; SATTERWAITE, E. **The impact of Mesa on System Design**. In Proceedings of the 4th International Conference on Software engeneering, pp 174-82. IEEE (1979)
[26]	CORNHILL, D. et al **Limitations of Ada for Real Time Scheduling**. Proc International Workshop on Real Time Ada Issues, ACM Ada Letters, pp. 33-9 (1987).
[27]	LEHOCZKY, J.P.; SHA, L.; STROSNIDER, J.K. - **Aperiodic Responsiveness in Hard Real-Time Environments**. Proc. of the IEEE Real-Time Systems Symp., pp. 262-270, 1987.
[28]	RAJKUMAR, R. **Synchronization in real-time systems:** A Priority Inheritance Approach. Kluwer Academic Publishers, 1991.
[29]	SHA, L.; RAJKUMAR, R.; LEHOCZKY, J. P. **Priority Inheritance Protocols:** An approach to real-time synchronization. IEEE Transactions on Computers, vol. 39, pp. 1175-1185, Sep. 1990.
[30]	BAKER, T. - **Stack-based scheduling of real-time processes**. Real-Time Systems Journal, 3(1), pp. 67-99, March, 1991.
[31]	RIBEIRO, J.R.P., SILVA, N.C., MORON, C.E. **A Visual Environment for the Development of Parallel Real-Time Programs**. Lecture Notes in Computer Science, v. 1388, p. 994-1014, 1998.
[32]	Internet Protocol. **DARPA Internet Program Protocol Specification:** Communication Layers, RFC 791, Setembro 1981.

Anexo A
Estrutura do Kernel SaoCarlOS

A.1 Arquivo kernel.c

A.1.1 Função Kernel_KernelInit

A.1.1.1 Descrição

Esta função inicializa as variáveis locais do kernel.

A.1.1.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.1.1.3 Protótipo:

```
void
Kernel_KernelInit()
```

A.1.2 Função Kernel_KernelPanic

A.1.2.1 Descrição

Esta função é chamada quando o Kernel atinge um estado ilegal. Ela deve terminar a execução do Kernel.

A.1.2.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.1.2.3 Protótipo:

```
void
Kernel_KernelPanic()
```

A.1.3 Função Kernel_KernelInitGreeting

A.1.3.1 Descrição

Esta função é chamada pela função Kernel_KernelStart de modo a mostrar a saudação inicial do sistema.

A.1.3.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.1.3.3 Protótipo

```
void
Kernel_KernelInitGreeting()
```

A.1.4 Função Kernel_KernelStart

A.1.4.1 Descrição

Esta função inicializa variáveis privadas do kernel e entra em uma estrutura de repetição infinita. Dentro desta estrutura de repetição as tarefas são executadas.

A.1.4.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.1.4.3 Protótipo

```
void
Kernel_KernelStart()
```

A.1.5 Função Kernel_KernelInterruptInit

A.1.5.1 Descrição

Esta função permite tratar um sinal de CTRL+C assim que ele chegar, de modo a finalizar a execução do kernel.

A.1.5.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.1.5.3 Protótipo

```
KernelStatus_T
Kernel_KernelInterruptInit()
```

A.1.6 Função Kernel_KernelExit

A.1.6.1 Descrição

Esta função libera recursos alocados pelo Sistema e finaliza a execução do Kernel.

A.1.6.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.1.6.3 Protótipo

```
void
Kernel_KernelExit()
```

A.2 Arquivo memory.c

A.2.1 Função Kernel_MemoryAllocate

A.2.1.1 Descrição

Esta função irá alocar memória com tamanho equivalente ao passado pela variável MemorySize.

A.2.1.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.2.1.3 Protótipo

```
void*
Kernel_MemoryAllocate(int MemorySize)
```

A.2.2 Função Kernel_MemoryRelease

A.2.2.1 Descrição

Esta função irá liberar a memória alocada referenciada pelo ponteiro para AllocatedMemory.

A.2.2.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.2.2.3 Protótipo

```
KernelStatus_T
Kernel_MemoryRelease(void* AllocatedMemory)
```

A.2.3 Função Kernel_MemoryCopyBlock

A.2.3.1 Descrição

Esta função irá copiar um bloco de memória de SourceBlock para DestinationBlock. A quantidade de informação a ser copiada deve ser definida em BlockSize.

A.2.3.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.2.3.3 Protótipo

```
KernelStatus_T
Kernel_MemoryCopyBlock(void* SourceBlock,
                       void* DestinationBlock,
                       int BlockSize)
```

A.2.4 Função Kernel_MemorySharedAllocate

A.2.4.1 Descrição

Esta função irá criar um bloco de memória compartilhada, permitindo comunicação interprocessos.

A.2.4.2 Pré Condição

SharedMemoryInstance está alocado.

A.2.4.3 Protótipo

```
void*
Kernel_MemorySharedAllocate(SharedMemory_T*
                                SharedMemoryInstance,
                            int SharedMemorySize)
```

A.2.5 Função Kernel_MemorySharedRelease

A.2.5.1 Descrição

Esta função libera um bloco de memória compartilhada.

A.2.5.2 Pré Condição

SharedMemoryInstance está alocado.

A.2.5.3 Protótipo

```
KernelStatus_T
Kernel_MemorySharedRelease(SharedMemory_T*
                               SharedMemoryInstance,
                           void* SharedMemory)
```

A.3 Arquivo network.c

A.3.1 Função Kernel_NetworkUDPCreate

A.3.1.1 Descrição

Esta função irá abrir uma porta lógica definida no argumento Port. Ela também ira permitir que uma tarefa receba e envie UDP PDU-Ts.

A.3.1.2 Pré Condição

A memória referenciada por NetworkUDP deve estar alocada.

A.3.1.3 Protótipo

```
KernelStatus_T
Kernel_NetworkUDPCreate(NetworkUDP_T* NetworkUDP,
                        int Port)
```

A.3.2 Função Kernel_NetworkUDPReceive

A.3.2.1 Descrição

Esta função irá permanecer bloqueada até o recebimento de um UDP PDU-T na porta designada por Kernel_NetworkUDPCreateServer.

A.3.2.2 Pré Condição

A função Kernel_NetworkUDPCreate deve ter sido executada anteriormente.

A.3.2.3 Protótipo

```
KernelStatus_T
Kernel_NetworkUDPReceive(NetworkUDP_T* NetworkUDP,
                         char* Message)
```

A.3.3 Função Kernel_NetworkUDPRespond

A.3.3.1 Descrição

Esta função responde com um UDP PDU-T com destino ao endereço de origem da ultima mensagem UDP recebida.

A.3.3.2 Pré Condição

A função Kernel_NetworkUDPCreate deve ter sido executada anteriormente.

A.3.3.3 Protótipo

```
KernelStatus_T
Kernel_NetworkUDPReceive(NetworkUDP_T* NetworkUDP,
                         char* Message)
```

A.3.4 Função Kernel_NetworkUDPSend

A.3.4.1 Descrição

Esta função ira enviar uma UDP PDU-T através da porta designada em Kernel_NetworkUDPCreate.

A.3.4.2 Pré Condição

A função Kernel_NetworkUDPCreate deve ter sido executada anteriormente.

A.3.4.3 Protótipo

```
KernelStatus_T
Kernel_NetworkUDPSend(NetworkUDP_T* NetworkUDP,
                      char* Message)
```

A.3.5 Função Kernel_NetworkUDPSetRemote

A.3.5.1 Descrição

Esta função configura o endereço da estação remota na estrutura NetworkUDP através do parâmetro RemoteStationName.

A.3.5.2 Pré Condição

A função Kernel_NetworkUDPCreate deve ter sido executada anteriormente.

A.3.5.3 Protótipo

```
KernelStatus_T
Kernel_NetworkUDPSetRemote(NetworkUDP_T* NetworkUDP,
                           char* RemoteStationName,
                           int RemoteStationPort)
```

A.3.6 Função Kernel_NetworkUDPTerminate

A.3.6.1 Descrição

Esta função ira fechar uma porta lógica designada a um socket anteriormente.

A.3.6.2 Pré Condição

A função Kernel_NetworkUDPCreate deve ter sido executada anteriormente.

A.3.6.3 Protótipo

```
KernelStatus_T
Kernel_NetworkTerminate(NetworkUDP_T* NetworkUDP)
```

A.3.7 Função Kernel_NetworkGetIP

A.3.7.1 Descrição

Esta função ira retornar o endereço IP da estação local.

A.3.7.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.3.7.3 Protótipo

```
char*
Kernel_NetworkGetIP()
```

A.4 Arquivo process.c

A.4.1 Função Kernel_ProcessGetKernelProcessID

A.4.1.1 Descrição

Esta função ira retornar a identificação do processo em execução.

A.4.1.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.4.1.3 Protótipo

```
ProcessID_T
Kernel_ProcessGetKernelProcessID()
```

A.4.2 Função Kernel_ProcessGetProcessID

A.4.2.1 Descrição

Esta função ira retornar a identificação do processo em execução.

A.4.2.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.4.2.3 Protótipo

```
ProcessID_T
Kernel_ProcessGetProcessID()
```

A.4.3 Função Kernel_ProcessGetPriority

A.4.3.1 Descrição

Esta função ira retornar a prioridade do processo em execução.

A.4.3.2 Pré Condição

Esta função deve ser chamada somente de processos de usuário.

A.4.3.3 Protótipo

```
Priority_T
Kernel_ProcessGetPriority()
```

A.4.4 Função Kernel_ProcessSetPriority

A.4.3.1 Descrição

Esta função irá enviar o evento KERNEL_EVENT_PROCESS_CHANGE_PRIORITY para o processo do kernel, causando a modificação da prioridade do processo alvo.

A.4.4.2 Pré Condição

Esta função deve ser chamada somente de processos de usuário.

A.4.4.3 Protótipo

```
void
Kernel_ProcessSetPriority(ProcessID_T ProcessID,
                          Priority_T Priority)
```

A.4.5 Função Kernel_ProcessTerminate

A.4.5.1 Descrição

Esta função ira enviar o evento KERNEL_EVENT_PROCESS_FINISH para o processo do kernel, causando a modificação do estado do processo em execução para TERMINATED e posteriormente a liberação das estruturas de processo do kernel.

A.4.5.2 Pré Condição

Esta função deve ser chamada somente de processos de usuário.

A.4.5.3 Protótipo

```
void
Kernel_ProcessTerminate()
```

A.4.6 Função Kernel_ProcessTerminateProcess

A.4.6.1 Descrição

Esta função ira enviar o evento KERNEL_EVENT_PROCESS_FINISH para o processo do kernel, causando a modificação do estado do processo alvo para TERMINATED e posteriormente a liberação das estruturas de processo do kernel.

A.4.6.2 Pré Condição

Esta função deve ser chamada somente de processos de usuário.

A.4.6.3 Protótipo

```
void
Kernel_ProcessTerminateProcess(ProcessID_T ProcessID)
```

A.4.7 Função Kernel_ProcessWait

A.4.7.1 Descrição

Esta função ira enviar o evento KERNEL_EVENT_PROCESS_BLOCK_STOPPED para o processo do kernel, causando a modificação do estado do processo em execução para BLOCKED.

A.4.7.2 Pré Condição

Esta função deve ser chamada somente de processos de usuário.

A.4.7.3 Protótipo

```
void
Kernel_ProcessWait()
```

A.4.8 Função Kernel_ProcessWaitProcess

A.4.8.1 Descrição

Esta função ira enviar o evento KERNEL_EVENT_PROCESS_BLOCK_STOPPED para o processo do kernel, causando a modificação do estado do processo alvo para BLOCKED.

A.4.8.2 Pré Condição

Esta função deve ser chamada somente de processos de usuário.

A.4.8.3 Protótipo

```
void
Kernel_ProcessWaitProcess(ProcessID_T ProcessID)
```

A.4.9 Função Kernel_ProcessYield

A.4.9.1 Descrição

Esta função ira enviar o evento KERNEL_EVENT_PROCESS_READY_STOPPED para o processo do kernel, causando a modificação do estado do processo em execução para READY.

A.4.9.2 Pré Condição

Esta função deve ser chamada somente de processos de usuário.

A.4.9.3 Protótipo

```
void
Kernel_ProcessYield()
```

A.4.10 Função Kernel_ProcessResume

A.4.10.1 Descrição

Esta função ira enviar o evento KERNEL_EVENT_PROCESS_READY_STOPPED para o processo do kernel, causando a modificação do estado do processo alvo para READY.

A.4.10.2 Pré Condição

Esta função deve ser chamada somente de processos de usuário.

A.4.10.3 Protótipo

```
void
Kernel_ProcessResume(ProcessID_T ProcessID)
```

A.4.11 Função Kernel_ProcessGetParentProcessID

A.4.11.1 Descrição

Esta função retorna o valor da identificação do processo criador do processo em execução.

A.4.11.2 Pré Condição

Esta função deve ser chamada somente de processos de usuário.

A.4.11.3 Protótipo

```
ProcessID_T
Kernel_ProcessGetParentProcessID()
```

A.4.12 Função Kernel_ProcessSearchProcessByID

A.4.12.1 Descrição

Esta função ira procurar por um processo dentro da estrutura de processos. Se a identificação do processo coincidir com a variável ProcessID, esta função irá retornar KERNEL_SUCCESS juntamente com o processo requisitado. Caso contrário, ela retornará KERNEL_FAIL.

A.4.12.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.4.12.3 Protótipo

```
KernelStatus_T
Kernel_ProcessSearchProcessByID(Process_T** Process,
                                ProcessID_T ProcessID)
```

A.4.13 Função Kernel_ProcessChangePriority

A.4.13.1 Descrição

Esta função modifica a prioridade dinâmica do processo alvo. Ela causa modificações na estrutura ProcessTable e também pode causar a preempção do processo em execução (caso a prioridade passada como parâmetro seja superior a do processo em execução).

A.4.13.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.4.13.3 Protótipo

```
KernelStatus_T
Kernel_ProcessChangePriority(Process_T* Process,
                             Priority_T NewPriority)
```

A.4.14 Função Kernel_ProcessBlock

A.4.14.1 Descrição

Esta função atribui o valor Blocked à variável ProcessState e chama a chamada de sistema kill, interrompendo a execução do processo correspondente.

A.4.14.2 Pré Condição

Esta função pode somente ser chamada pelo Kernel. Para bloquear um processo a partir de um processo de usuário use o evento KERNEL_EVENT_PROCESS_BLOCK_STOPPED.

A.4.14.3 Protótipo

```
KernelStatus_T
Kernel_ProcessBlock(Process_T* Process)
```

A.4.15 Função Kernel_ProcessStop

A.4.15.1 Descrição

Esta função atribui o valor Ready à variável ProcessState e chama a chamada de sistema kill, interrompendo a execução do processo correspondente.

A.4.15.2 Pré Condição

Esta função pode somente ser chamada pelo Kernel. Para bloquear um processo a partir de um processo de usuário use o evento KERNEL_EVENT_PROCESS_STOP.

A.4.15.3 Protótipo

```
KernelStatus_T
Kernel_ProcessStop(Process_T* Process)
```

A.4.16 Função Kernel_ProcessContinue

A.4.16.1 Descrição

Esta função atribuir o valor Running à variável ProcessState e chama a chamada de sistema kill, interrompendo a execução do processo correspondente.

A.4.16.2 Pré Condição

Esta função pode somente ser chamada pelo Kernel. Para bloquear um processo a partir de um processo de usuário use o evento KERNEL_EVENT_PROCESS_RESUME.

A.4.16.3 Protótipo

```
KernelStatus_T
Kernel_ProcessContinue(Process_T* Process)
```

A.4.17 Função Kernel_ProcessFinish

A.4.17.1 Descrição

Esta função remove o processo referenciado por Process da estrutura de processos. Ela também executa kill matando o processo e libera a memória alocada por ele.

A.4.17.2 Pré Condição

Esta função pode somente ser chamada pelo Kernel. Para bloquear um processo a partir de um processo de usuário use o evento KERNEL_EVENT_PROCESS_TERMINATE.

A.4.17.3 Protótipo

```
KernelStatus_T
Kernel_ProcessFinish(Process_T* Process)
```

A.4.18 Função Kernel_ProcessRunningFinish

A.4.18.1 Descrição

Esta função remove o processo em execução da estrutura de processos. Ela também executa kill matando o processo e libera a memória alocada por ele.

A.4.18.2 Pré Condição

Esta função pode somente ser chamada pelo Kernel. Para bloquear um processo a partir de um processo de usuário use o evento KERNEL_EVENT_PROCESS_TERMINATE.

A.4.18.3 Protótipo

```
void
Kernel_ProcessRunningFinish()
```

A.4.19 Função Kernel_ProcessFinishAll

A.4.19.1 Descrição

Esta função remove todos processos da estrutura de processos. Ela também executa kill matando os processos e libera a memória alocada por eles.

A.4.19.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.4.19.3 Protótipo

```
KernelStatus_T
Kernel_ProcessFinishAll()
```

A.4.20 Função Kernel_ProcessSchedule

A.4.20.1 Descrição

Esta função procura pelo próximo processo pronto para execução e o coloca no estado running. Esta função implementa os algoritmos Round Robin e Escalonamento por Prioridade, dependendo da variável de compilação SCHEDULE.

A.4.20.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.4.20.3 Protótipo

```
KernelStatus_T
Kernel_ProcessSchedule()
```

A.4.21 Função Kernel_ProcessCreate

A.4.21.1 Descrição

Esta função inclui um novo processo na estrutura de processos. Note que quando executado a partir do processo do kernel (durante a inicialização do sistema) o perfil do processo deve ser obrigatoriamente passado através da variável Process. Caso contrário, quando chamando a função através de um processo de usuário, o parâmetro Process não é necessário, podendo ser substituído por KERNEL_NULL. A implementação da tarefa é passada pela variável ProcessImplementation.

A.4.21.2 Pré Condição

Ao ser chamada pelo processo do kernel, a variável Process deve ser obrigatoriamente passado como parâmetro. O ponteiro para função Task deve sempre ser passada.

A.4.21.3 Protótipo

```
KernelStatus_T
Kernel_ProcessCreate(Process_T *Process,
                     void (*ProcessImplementation)())
```

A.4.22 Função Kernel_ProcessKernelCreate

A.4.22.1 Descrição

Esta função inclui um novo processo na estrutura de processos. Note que o perfil do processo deve ser obrigatoriamente passado através da variável Process. A implementação da tarefa é passada pela variável ProcessImplementation.

A.4.22.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.4.22.3 Protótipo

```
KernelStatus_T
Kernel_ProcessKernelCreate(Process_T *Process,
                           void
                           (*ProcessImplementation)())
```

A.4.23 Função Kernel_ProcessUserCreate

A.4.23.1 Descrição

Esta função inclui um novo processo na estrutura de processos. A implementação da tarefa é passada pela variável ProcessImplementation.

A.4.23.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.4.23.3 Protótipo

```
KernelStatus_T
Kernel_ProcessUserCreate(Process_T *Process,
                         void
                         (*ProcessImplementation)())
```

A.4.24 Função Kernel_ProcessCreateHandler

A.4.24.1 Descrição

Esta função trata o evento KERNEL_EVENT_PROCESS_CREATE e inclui um novo processo nas estruturas do kernel. Note que os atributos de tempo

real do novo processo irá herdar os atributos de tempo real do processo que o criou.

A.4.24.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.4.24.3 Protótipo

```
KernelStatus_T
Kernel_ProcessCreateHandler(Process_T *ProcessOwner,
                            ProcessID_T ProcessID)
```

A.5 Arquivo semaphore.c

A.5.1 Função Kernel_SemaphoreCreate

A.5.1.1 Descrição

Esta função cria um certo número de semáforos definidos no parâmetro TotalSemaphores.

A.5.1.2 Pré Condição

A instancia para Semaphore_T deve estar alocada.

A.5.1.3 Protótipo

```
KernelStatus_T
Kernel_SemaphoreCreate(Semaphore_T* SemaphoreInstance,
                       int TotalSemaphores)
```

A.5.2 Função Kernel_SemaphoreInit

A.5.2.1 Descrição

Esta função inicializa um semáforo com o valor passado em SemaphoreValue. O parâmetro SemaphoreNumber indica qual semáforo deve ser inicializado. O valor de SemaphoreNumber pode ser qualquer número no intervalo:
[0,TotalSemaphores) onde TotalSemaphores foi passado como parâmetro de Kernel_SemaphoreCreate().

A.5.2.2 Pré Condição

A função Kernel_SemaphoreCreate() foi chamada e retornou sucesso.

A.5.2.3 Protótipo

```
KernelStatus_T
Kernel_SemaphoreInit(Semaphore_T* SemaphoreInstance,
                     int SemaphoreNumber,
                     int SemaphoreValue)
```

A.5.3 Função Kernel_SemaphoreDown

A.5.3.1 Descrição

Esta função decrementa o valor do semáforo com o valor definido no parâmetro OperationValue. O parâmetro SemaphoreNumber indica qual semáforo deve ser decrementado. SemaphoreNumber pode assumir qualquer valor dentro do intervalo:
[0,TotalSemaphores) onde TotalSemaphores foi passado como parâmetro de Kernel_SemaphoreCreate().

A.5.3.2 Pré Condição

A função Kernel_SemaphoreCreate() foi chamada e retornou sucesso.

A.5.3.3 Protótipo

```
KernelStatus_T
Kernel_SemaphoreDown(Semaphore_T* SemaphoreInstance,
                     int SemaphoreNumber,
                     int OperationValue)
```

A.5.4 Função Kernel_SemaphoreUp

A.5.4.1 Descrição

Esta função incrementa o valor do semáforo com o valor definido no parâmetro OperationValue. O parâmetro SemaphoreNumber indica qual

semáforo deve ser incrementado. SemaphoreNumber pode assumir qualquer valor dentro do intervalo:
[0,TotalSemaphores) onde TotalSemaphores foi passado como parâmetro de Kernel_SemaphoreCreate().

A.5.4.2 Pré Condição

A função Kernel_SemaphoreCreate() foi chamada e retornou sucesso.

A.5.4.3 Protótipo

```
KernelStatus_T
Kernel_SemaphoreUp(Semaphore_T* SemaphoreInstance,
                   int SemaphoreNumber,
                   int OperationValue)
```

A.5.5 Função Kernel_SemaphoreTerminate

A.5.5.1 Descrição

Esta função libera recursos e estruturas associadas ao semáforo.

A.5.5.2 Pré Condição

A função Kernel_SemaphoreCreate() foi chamada e retornou sucesso.

A.5.5.3 Protótipo

```
KernelStatus_T
Kernel_SemaphoreTerminate(Semaphore_T*
                          SemaphoreInstance)
```

A.6 Arquivo time.c

A.6.1 Função Kernel_TimeSetTimer

A.6.1.1 Descrição

Esta função permite o kernel criar um timer para uma determinada tarefa. Tal timer permite que o tempo de execução no processador seja concedido a um outro processo.

A.6.1.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.6.1.3 Protótipo

```
KernelStatus_T
Kernel_TimeSetTimer(int SecondsSleepTime,
                    long NanoSecondsSleepTime)
```

A.6.2 Função Kernel_TimeSleep

A.6.2.1 Descrição

Esta função permite que um processo fique bloqueado por um tempo determinado pelo parâmetro Time.

A.6.2.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.6.2.3 Protótipo

```
KernelStatus_T
Kernel_TimeSleep(int SecondsSleepTime)
```

A.6.3 Função Kernel_TimeGetTime

A.6.3.1 Descrição

Esta função retorna o tempo do Kernel. Isto pode permitir ao usuário calcular o tempo gasto na execução de uma certa parte do código (de modo a coletar benchmarks).

A.6.3.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.6.3.3 Protótipo

```
KernelTime_T
Kernel_TimeGetTime()
```

A.7 Arquivo event.c

A.7.1 Função Kernel_EventHandlerInit

A.7.1.1 Descrição

Esta função inicia a funcionalidade de manipulador de Eventos. Após isto, é permitido que eventos sejam enviados de processos do Usuário para processos do Kernel.

A.7.1.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.7.1.3 Protótipo

```
KernelStatus_T
Kernel_EventHandlerInit()
```

A.7.2 Função Kernel_EventSend

A.7.2.1 Descrição

Esta função envia um evento ao processo do Kernel contendo código do evento e informações para o processo destino.

A.7.2.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.7.2.3 Protótipo

```
void
Kernel_EventSend(EventID_T EventID,
                 ProcessID_T ProcessID)
```

A.7.3 Função Kernel_EventHandler

A.7.3.1 Descrição

Esta função decodifica e executa o manipulador correspondente ao evento enviado por um processo de Usuário ou por um processo de Kernel.

A.7.3.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.7.3.3 Protótipo

```
void
Kernel_EventHandler()
```

A.7.4 Função Kernel_EventIncrement

A.7.4.1 Descrição

Esta função incrementa o valor de TotalEventsToHandle permitindo ao kernel identificar quantas vezes chamar Kernel_EventHandler() .

A.7.4.2 Pré Condição

Esta função deve apenas ser chamada pelo Kernel. Não existe uma outra chamada que executa a mesma operação para um processo do Usuário.

A.7.4.3 Protótipo

```
void
Kernel_EventIncrement()
```

A.8 Arquivo util.c

A.8.1 Função Kernel_UtilShow

A.8.1.1 Descrição

Esta função é utilizada para mostrar recursos de texto na saída padrão.

A.8.1.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.8.1.3 Protótipo

```
void
Kernel_UtilShow(char* TextResource)
```

A.8.2 Função Kernel_UtilIntToStr

A.8.2.1 Descrição

Esta função é utilizada para converter um inteiro em uma cadeia de caracteres contendo tal valor.

A.8.2.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.8.2.3 Protótipo

```
char* Kernel_UtilIntToStr(int Value)
```

A.8.3 Função Kernel_UtilStrToIntFreeStr

A.8.3.1 Descrição

Esta função é utilizada para converter um inteiro contido em uma cadeia de caracteres para um valor inteiro. Após a conversão a memória alocada para a cadeia de caracteres é liberada.

A.8.3.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.8.3.3 Protótipo

```
int Kernel_UtilStrToIntFreeStr(char* String)
```

A.8.4 Função Kernel_UtilGetToken

A.8.4.1 Descrição

Esta função recupera a cadeia de caracteres do início de *Text até a primeira ocorrência de Delimiter. Tal cadeia de caracteres mais o delimitador, é removido de *Text.

A.8.4.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.8.4.3 Protótipo

```
char*
Kernel_UtilGetToken(char** Text, char Delimiter)
```

A.8.5 Função Kernel_UtilDebug

A.8.5.1 Descrição

Esta função é utilizada para criar mensagens de log pelo kernel.

A.8.5.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.8.5.3 Protótipo

```
void
Kernel_UtilDebug(char* TextResource,int Value)
```

A.9 Arquivo naming.c

A.9.1 Função Kernel_UtilDebug

A.9.1.1 Descrição

Esta função inicia a execução do Servidor de Nomes.

A.9.1.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.9.1.3 Protótipo

```
KernelStatus_T
NamingServer_Init()
```

A.10 Arquivo gap.c

A.10.1 Função GAP_AddElement

A.10.1.1 Descrição

Esta função adiciona um elemento na tabela de pontos de acesso globais (GAP – Global Access Point).

A.10.1.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.10.1.3 Protótipo

```
GAP_AddElement(GlobalAccessPoint_T GlobalAccessPoint,
               char* StationAddress,
               int StationPort)
```

A.10.2 Função GAP_UpdateElement

A.10.2.1 Descrição

Esta função atualiza as informações de um elemento na tabela de pontos de acesso globais (GAP – Global Access Point).

A.10.2.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.10.2.3 Protótipo

```
GAP_UpdateElement(GlobalAccessPoint_T GlobalAccessPoint,
                  char* StationAddress,
                  int StationPort)
```

A.10.3 Função GAP_RemoveElement

A.10.3.1 Descrição

Esta função remove um elemento na tabela de pontos de acesso globais (GAP – Global Access Point).

A.10.3.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.10.3.3 Protótipo

```
KernelStatus_T
GAP_RemoveElement(GlobalAccessPoint_T GlobalAccessPoint)
```

A.10.4 Função GAP_GetElement

A.10.4.1 Descrição

Esta função recupera as informações de um elemento na tabela de pontos de acesso globais (GAP – Global Access Point).

A.10.4.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.10.4.3 Protótipo

```
KernelStatus_T
GAP_GetElement(GlobalAccessPoint_T GlobalAccessPoint,
               char** StationAddress,
               int* StationPort)
```

A.10.4 Função GAP_GetElementTTL

A.10.4.1 Descrição

Esta função avalia o tempo de vida da entrada da tabela antes de recuperar as informações de um elemento na tabela de pontos de acesso globais (GAP – Global Access Point). Caso o tempo de vida expirou a função retornará KERNEL_FAIL.

A.10.4.2 Pré Condição

Não existe nenhuma precondição referente àchamada desta função.

A.10.4.3 Protótipo

```
KernelStatus_T
GAP_GetElementTTL(GlobalAccessPoint_T GlobalAccessPoint,
                  char** StationAddress,
                  int* StationPort)
```

A.11 Arquivo message.c

A.11.1 Função Kernel_MessageRTPrefetch

A.11.1.1 Descrição

Caso ainda não exista uma entrada no cache local, esta função gera uma requisição para o Servidor de Nomes pedindo a resolução do endereço de TargetAccessPoint. Após a resposta do servidor, tal endereço é armazenado no cache para futuras consultas.

A.11.1.2 Pré Condição

Esta função deve ser chamada sempre após `Kernel_KernelInit()`.

A.11.1.3 Protótipo

```
KernelStatus_T
Kernel_MessageRTPrefetch(GlobalAccessPoint_T
                         TargetAccessPoint)
```

A.11.2 Função Kernel_MessageRTSend

A.11.2.1 Descrição

Caso ainda não exista uma entrada no cache local, esta função gera uma requisição para o Servidor de Nomes pedindo a resolução do endereço de TargetAccessPoint. Após a resposta do servidor, tal endereço é armazenado no cache para futuras consultas.

Após o conhecimento do endereço destino, uma mensagem de tempo real é enviada através da rede de comunicação. Tal mensagem contem o PriorityCeiling, equivalente a prioridade dinâmica do processo emissor.

A.11.2.2 Pré Condição

Esta função deve ser chamada sempre após `Kernel_MessageRTCreate()`.

A.11.2.3 Protótipo

```
KernelStatus_T
Kernel_MessageRTSend(RTMessage_T* MessageInstance,
                     char* Message,
                     GlobalAccessPoint_T
                        TargetAccessPoint)
```

A.11.3 Função Kernel_MessageRTReceive

A.11.3.1 Descrição

Esta função bloqueia a execução do processo receptor até o recebimento de uma mensagem. Dado seu recebimento, e caso a prioridade dinâmica do processo for inferior ao priority ceiling da mensagem, a prioridade dinâmica herdará o valor do priority ceiling.

A.11.3.2 Pré Condição

Esta função deve ser chamada sempre após Kernel_KernelInit().

A.11.3.3 Protótipo

```
KernelStatus_T
Kernel_MessageRTReceive(RTMessage_T* MessageInstance,
                        char** Message,
                        GlobalAccessPoint_T*
                           SenderAccessPoint)
```

A.11.4 Função Kernel_MessageRTInit

A.11.4.1 Descrição

Esta função inicializa todas as estruturas de troca de mensagens, assim como faz o registro do Ponto de Acesso Global no Servidor de Nomes. A partir de então este processo está apto ao recebimento de mensagens de tempo real.

A.11.4.2 Pré Condição

A instancia de MessageRT está devidamente alocada.

A.11.4.3 Protótipo

```
KernelStatus_T
Kernel_MessageRTInit(RTMessage_T* MessageInstance,
                     GlobalAccessPoint_T
                        GlobalAccessPoint)
```

A.11.5 Função Kernel_MessageRTFinish

A.11.5.1 Descrição

Esta função gera uma requisição de desregistro do Ponto de Acesso Global para o Servidor de Nomes. Ela também volta a prioridade do processo ao valor do maior priority ceiling utilizado pelas outras instancias de MessageRT.

A.11.5.2 Pré Condição

Esta função deve ser chamada sempre após Kernel_KernelInit().

A.11.5.3 Protótipo

```
KernelStatus_T
Kernel_MessageRTFinish(RTMessage_T* MessageInstance)
```

A.12 Arquivo resource.c

A.12.1 Função Kernel_ResourceCreate

A.12.1.1 Descrição

Esta função cria um Recurso. Tal recurso pode ser utilizado para fazer o controle de acesso a um dispositivo compartilhado através de bloqueio e desbloqueio.

A.12.1.2 Pré Condição

A instância para o recurso deve estar alocada.

A.12.1.3 Protótipo

```
KernelStatus_T
Kernel_ResourceCreate(Resource_T* ResourceInstance)
```

A.12.2 Função Kernel_ResourceLock

A.12.2.1 Descrição

Esta função é utilizada para fazer o bloqueio de um dispositivo compartilhado.

A.12.2.2 Pré Condição

A função Kernel_ResourceCreate() deve ter sido executada anteriormente e retornado KERNEL_SUCCESS.

A.12.2.3 Protótipo

```
KernelStatus_T
Kernel_ResourceLock(Resource_T* ResourceInstance)
```

A.12.3 Função Kernel_ResourceUnlock

A.12.3.1 Descrição

Esta função é utilizada para fazer o desbloqueio de um dispositivo compartilhado.

A.12.3.2 Pré Condição

A função Kernel_ResourceCreate() deve ter sido executada anteriormente e retornado KERNEL_SUCCESS.

A.12.3.3 Protótipo

```
KernelStatus_T
Kernel_ResourceUnock(Resource_T* ResourceInstance)
```

A.12.4 Função Kernel_ResourceTerminate

A.12.4.1 Descrição

Esta função é utilizada para liberar toda memória alocada e utilizada pelo recurso.

A.12.4.2 Pré Condição

A função Kernel_ResourceCreate() deve ter sido executada anteriormente e retornado KERNEL_SUCCESS.

A.12.4.3 Protótipo

```
KernelStatus_T
Kernel_ResourceTerminate(Resource_T* ResourceInstance)
```

A.13 Arquivo mailbox.c

A.13.1 Função Kernel_MailboxCreate

A.13.1.1 Descrição

Esta função cria um Mailbox. Tal Mailbox é composto de uma lista circular em forma de Fila (First In First Out). A partir dos parâmetros é possível determinar o número de blocos e o tamanho de cada bloco de memória.

A.13.1.2 Pré Condição

A instância para o Mailbox deve estar alocada.

A.13.1.3 Protótipo

```
KernelStatus_T
Kernel_MailboxCreate(Mailbox_T* MailboxInstance,
                     int BlockSize,
                     int TotalBlocks)
```

A.13.2 Função Kernel_MailboxRead

A.13.2.1 Descrição

Esta função é utilizada para fazer a leitura de um Mailbox. O parâmetro ReadBuffer deve estar alocado com o tamanho definido no tamanho do bloco de memória.

A.13.2.2 Pré Condição

A função Kernel_MailboxCreate() deve ter sido executada anteriormente e retornado KERNEL_SUCCESS.

A.13.2.3 Protótipo

```
KernelStatus_T
Kernel_MailboxRead(Mailbox_T* MailboxInstance,
                   char* ReadBuffer)
```

A.13.2 Função Kernel_MailboxWrite

A.13.2.1 Descrição

Esta função é utilizada para fazer a escrita em um Mailbox. O parâmetro WriteBuffer deve estar alocado com o tamanho definido no tamanho do bloco de memória.

A.13.2.2 Pré Condição

A função Kernel_MailboxCreate() deve ter sido executada anteriormente e retornado KERNEL_SUCCESS.

A.13.2.3 Protótipo

```
KernelStatus_T
Kernel_MailboxWrite(Mailbox_T* MailboxInstance,
                    char* ReadBuffer)
```

A.13.2 Função Kernel_MailboxTerminate

A.13.2.1 Descrição

Esta função é utilizada para liberar a memória alocada pelo Mailbox, assim como desalocar todas as estruturas utilizadas por ele.

A.13.2.2 Pré Condição

A função Kernel_MailboxCreate() deve ter sido executada anteriormente e retornado KERNEL_SUCCESS.

A.13.2.3 Protótipo

```
KernelStatus_T
Kernel_MailboxTerminate(Mailbox_T* MailboxInstance)
```

Anexo B

Tempos de Trocas de Mensagem

Amostra (Mensagem)	Nomes Globais (milisegundos)	Servidor de Nomes (milisegundos)	Servidor de Nomes Cache Local (milisegundos)	Servidor de Nomes Cache Local Prefetching (milisegundos)
1	9,866	30,220	29,989	9,938
2	9,980	30,589	9,866	9,773
3	9,858	30,224	10,010	9,738
4	10,061	29,577	10,222	9,900
5	10,017	30,319	9,933	9,889
6	10,081	29,857	10,235	9,981
7	10,163	29,789	10,130	10,026
8	10,335	30,051	10,058	9,820
9	9,983	30,317	9,847	9,912
10	10,245	30,241	10,050	10,091
11	10,126	30,481	9,964	10,022
12	9,850	29,324	10,060	9,993
13	9,886	30,052	10,026	10,075
14	9,876	30,034	10,017	10,199
15	9,906	30,501	9,854	10,014
16	10,070	30,084	9,892	9,963
17	9,990	30,141	10,055	10,020
18	9,750	29,985	10,197	9,972
19	9,921	30,511	10,010	10,009
20	10,084	29,996	9,772	9,915
21	9,991	29,192	10,093	9,931
22	10,104	30,590	9,981	10,243
23	9,850	30,356	9,935	9,926
24	9,986	29,986	10,103	10,018
25	10,049	29,989	9,997	10,242
26	9,977	29,887	9,957	10,027
27	10,205	29,607	10,163	9,924
28	9,907	30,292	10,052	10,199
29	9,954	30,557	9,959	10,061
30	9,960	30,384	10,080	10,113
31	9,933	30,221	9,933	10,061
32	10,145	29,723	10,032	10,299
33	9,964	29,415	10,022	10,158
34	10,032	29,722	9,958	9,980
35	10,182	30,318	10,291	10,017
36	9,996	29,789	9,993	9,901
37	10,098	29,879	9,923	9,974
38	9,904	29,591	9,965	10,022
39	9,799	29,837	10,065	10,047

40	10,124	29,601	10,144	10,185
41	9,932	30,705	9,864	10,072
42	9,909	29,862	10,035	10,117
43	9,847	29,718	10,137	9,925
44	10,096	30,166	10,043	10,091
45	10,014	29,750	9,832	9,899
46	10,052	30,084	10,102	10,099
47	9,801	29,873	10,052	10,125
48	9,907	29,736	10,062	9,944
49	10,209	29,901	10,034	10,146
50	10,026	30,290	10,031	10,121
51	10,076	29,805	10,059	9,801
52	9,961	30,635	10,221	10,165
53	9,705	30,567	10,258	10,177
54	9,994	30,201	10,170	10,319
55	10,068	29,523	9,976	9,899
56	10,084	30,749	10,194	9,809
57	10,008	29,866	9,974	9,959
58	10,013	29,445	10,029	10,297
59	10,169	29,987	9,780	9,913
60	10,020	30,298	9,900	9,935
61	10,076	30,025	10,080	9,975
62	10,026	30,087	9,948	10,013
63	10,034	30,146	9,924	10,135
64	10,180	30,361	10,049	9,968
65	10,021	30,359	10,057	9,828
66	10,125	29,883	10,112	10,113
67	10,036	30,041	9,949	10,076
68	10,244	29,322	10,001	10,058
69	10,188	30,709	10,207	10,119
70	10,100	30,019	9,716	9,879
71	9,918	30,041	10,081	10,043
72	10,053	30,987	10,139	9,794
73	10,113	30,131	9,963	9,973
74	10,039	29,861	10,153	10,095
75	10,048	30,020	9,993	9,898
76	9,895	30,356	9,815	10,060
77	10,125	30,476	9,998	10,022
78	10,038	29,770	10,309	9,961
79	10,151	30,124	9,895	9,963
80	9,956	30,233	10,136	10,025
81	9,911	30,035	10,174	10,020
82	10,029	30,799	9,894	9,935
83	9,941	29,701	10,189	9,974
84	10,253	30,811	10,014	10,119
85	10,097	29,615	9,886	10,232
86	10,292	30,401	10,137	10,186

87	9,877	30,135	10,114	10,092
88	9,974	30,151	9,918	10,045
89	10,053	30,351	9,919	9,955
90	10,108	30,289	10,045	10,072
91	9,956	29,763	9,828	10,014
92	10,133	30,422	10,173	10,174
93	10,070	29,493	9,930	10,060
94	10,049	29,616	10,021	9,857
95	10,193	30,228	10,123	10,078
96	10,185	30,156	10,068	9,876
97	10,028	29,642	9,927	10,139
98	9,941	29,577	10,124	9,960
99	10,105	30,071	9,970	10,007
100	10,119	30,162	10,023	10,108
101	9,875	30,395	9,923	10,236
102	9,813	29,914	10,091	10,149