UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E CIÊNCIA DA COMPUTAÇÃO
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

# Camada de Redirecionamento: Um novo paradigma para a análise de Sistemas Distribuídos

Carlos Frederico Marcelo da Cunha Cavalcanti[1]

Orientador: Hans Kurt Edmund Liesenberg

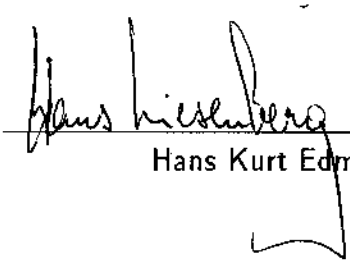
29 de abril de 1993

[a]e-mail: cfmcc@dcc.unicamp.br

# Camada de Redirecionamento: Um novo paradigma para a análise de Sistemas Distribuídos

Este exemplar corresponde a redação final da tese devidamente corrigida pelo Sr. Carlos Frederico Marcelo da Cunha Cavalcanti e aprovada pela Comissão Julgadora.

Campinas, 29 de abril de 1993.

Prof. Dr. 

Hans Kurt Edmund Liesenberg

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do Título de MESTRE em Ciência da Computação.

Carlos Frederico Marcelo da Cunha Cavalcanti

# Camada de Redirecionamento: Um novo paradigma para a análise de Sistemas Distribuídos

Dissertação apresentada em 29 de abril de 1993.

**Banca Examinadora:**

Dr. Hans Edmund Liesenberg – UNICAMP / DCC – Orientador

Dr. Maurício Ferreira Magalhães – UNICAMP / FEE

Dr. Ricardo de Oliveira Anido – UNICAMP / DCC

À minha mãe, à minha avó Lina e a Cleide.

# Agradecimentos

À minha mãe, pela confiança, dedicação, fé e amor que apesar de sempre presentes em toda a minha vida, foram decisivos durante este curso.

À minha avó, que com seu amor e ternura me apoiou e fortaleceu nesta importante etapa da minha vida.

À minha querida Cleide, que apoiou, incentivou e compartilhou todos os momentos deste curso de mestrado.

Ao Sr. Nilo Augusto Bretas, pelo carinho e apoio constante.

A Hans Edmund Liesenberg exemplo de professor e orientador, pelo apoio e confiança.

Aos amigos Carlos Salles Lambert, João Hermes Clerice e Paulo César Seron pelo acolhimento e amizade.

A Fernando Mascarenhas S. de Assis e a José Roberto de Oliveira por terem acreditado em meu trabalho.

Aos amigos da Pontíficia Universidade Católica de Minas Gerais, Attenister T. Rego, Luis P. Wilke Boratto e Nilson Figueiredo Filho, pelo apoio inicial e incentivo constante.

À Fundação Centro Tecnológico de Minas Gerais, CETEC-MG e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico, CNPq pelo apoio financeiro.

Aos meus familiares, amigos, professores e colegas que me incentivaram durante o curso de mestrado.

Às pessoas que intercederam a Deus por este seu filho.

A Deus, Pai de infinita bondade, por este curso de mestrado carinhosamente preparado para mim.

*"Computer science is not only a study of a basic theory, and it is not just the bussiness of making things happen. It's actually a study of how things happen."*

*Robin Milner [Frenkel93]*

# Abstract

A multicomputer system has some characteristics which, when well explored, allow us to get real throughtput and funcionality that aren't found in other systems, or even in each individual computer. In order to explore these characteristics it is necessary to integrate, in an adequate manner, all the single resources in only one resource, now representing the system. So, the integration of individual computers will produce in a gain if a real cooperation among them is established. The greater the engagement of these elements - hardware, operating systems, languagens and support tools - with the multicomputer system, the greater will be the gain of throughtput and funcionality of the system.

This dissertation characterizes these mechanisms (agents) whose aim is to make the cooperation of processors in a multicomputer system possible via code distribution. In order to achive this goal, an abstract entity and its computacional model were idealized which establishes a new paradigm for the analysis of distributed systems. Several multicomputer systems are discussed and their code distribution analysed under the proposed redirection layer paradigm.

# Sumário

Um sistema multicomputacional possui características que, quando bem exploradas, permitem obter um ganho real em desempenho e funcionalidade não encontrados em outros sistemas, ou mesmo em cada computador individualmente. Para explorar estas características é necessário integrar, de uma forma adequada, todos os recursos individuais em um recurso único, agora representados pelo sistema. Assim, a interligação de todos os computadores refletirá em ganho caso haja uma efetiva cooperação entre eles. Quanto maior o comprometimento dos elementos que participam desta cooperação - *hardware*, sistema operacional, linguagens e ferramentas de apoio - com o sistema multicomputacional, maior será o ganho de desempenho e funcionalidade do sistema.

Esta dissertação caracteriza estes mecanismos (agentes) que possuem o objetivo de viabilizar a cooperação dos processadores em um sistema multicomputacional através da distribuição de código. Para que isto fosse possível foi idealizado uma entidade abstrata denominada camada de redirecionamento. A camada de redirecionamento juntamente com seu modelo computacional constituem um novo paradigma para a análise de sistemas distribuídos. Vários sistemas multicomputacionais são discutidos analisando a distribuição de código sob a óptica da camada de redirecionamento.

# Conteúdo

# Lista de Figuras

# Capítulo 1

# Introdução

Estas duas últimas décadas foram caracterizadas por grandes atividades científicas visando explorar eficientemente o processamento paralelo. Tais atividades ocorreram praticamente em todo o espectro da ciência da computação, desde arquitetura de computadores, representada principalmente pelas máquinas de fluxo de dados e processadores vetoriais, passando por linguagens de programação, sistemas operacionais e chegando às questões mais teóricas da matemática computacional. Tais esforços estão disponíveis hoje em uma ampla gama de produtos, que direta ou indiretamente mudaram o conceito de computação puramente *von-Neuman*. Concomitantemente, os computadores tornaram-se cada vez mais poderosos e baratos, tendência que se confirma até hoje, viabilizando a interligação de vários computadores entre si através de uma rede de comunicação de alta velocidade.

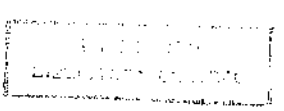
Explora-se o paralelismo de um sistema computacional composto por vários processadores interconectados através da distribuição de tarefas entre os diversos processadores de tal forma que cada processador realize parte do processamento total. Uma aplicação que explore este tipo de sistema computacional, agora representado pelo conjunto de processadores, pode ter objetivos bem distintos do aumento da capacidade de processamento, como por exemplo, prover a aplicação de tolêrancia a falhas.

Um conjunto de computadores distintos interconectados por uma rede de comunicação é um caso típico de um sistema computacional composto por vários processadores. O sistema composto por uma arquitetura de vários computadores que só possuem uma rede de comunicação para a sincronização e comunicação entre eles será denominado aqui **sistema multicomputacional.**

Um sistema multicomputacional possui características que, quando bem exploradas, permitem obter um ganho real em desempenho e funcionalidade não encontrados em outros sistemas, ou mesmo em cada computador separado. Para explorar estas características é necessário integrar, de uma forma adequada, todos os recursos individuais em um recurso único, agora representados pelo sistema. Assim, a interligação de todos os computadores refletirá em ganho caso haja uma efetiva cooperação entre eles. Quanto maior o comprometimento dos elementos que participam desta cooperação - *hardware*, sistema operacional, linguagens e ferramentas de apoio - com o sistema multicomputacional, maior será o ganho de desempenho e funcionalidade do sistema.

Esta dissertação enfoca principalmente os mecanismos (agentes) que possuem o objetivo de viabilizar a cooperação dos processadores em um sistema multicomputacional





através da distribuição de código entre eles. Compreender tais mecanismos, definir seus inter-relacionamentos e suas atuações possibilita estabelecer as características de um sistema multicomputacional, seu custo e o seu desempenho. Para que isso fosse possível, foi idealizado uma entidade abstrata, definida no capítulo 3, denominada **camada de redirecionamento**. Através da camada de redirecionamento iremos identificar e analisar diversos elementos responsáveis pela efetiva cooperação entre os diversos processadores do sistema. Também foi definido, no mesmo capítulo, um modelo computacional que servirá de referência em toda a nossa dissertação. A camada de redirecionamento juntamente com seu modelo computacional constituem um novo paradigma para a análise de sistemas distribuídos.

Inúmeras questões podem ser tratadas quando se explora um sistema multicomputacional sob a ótica da camada de redirecionamento e do modelo computacional propostos nesta dissertação. Cada questão deve ser orientada aos potenciais ganhos reais de desempenho e funcionalidade, como distribuição de dados entre os vários computadores do sistema, tolerância a falhas, distribuição de código, etc. Esta dissertação enfatiza uma questão fundamental no estudo de sistemas distribuídos que é a distribuição de código no sistema.

Não se objetiva resumir toda a discussão de um tema vasto e sedutor a um determinado sistema operacional ou a desenvolver código para uma determinada máquina. Objetiva-se, de outra forma, dar uma visão global e atual de várias alternativas encontradas para extrair e prover acesso, de uma forma adequada, aos recursos de uma sistema multicomputacional no que diz respeito ao código. Para atingir tal objetivo, o texto foi divido da seguinte forma: com o propósito de uniformizar a nomeclatura usada e enfatizar as particularidades de um sistema multicomputacional frente a outros sistemas que exploram paralelismo e/ou concorrência, iremos rever, no **capítulo 2**, alguns aspectos básicos na discussão de processamento concorrente e distribuído; no **capítulo 3** é apresentado o modelo computacional usado em toda a dissertação e a camada de redirecionamento; no **capítulo 4** é apresentado o redirecionamento forte pela função $f_L$, que consiste em caracterizar e exemplificar os mecanismos de redirecionamento implementados a partir da geração de código; no **capítulo 5** é apresentado o redirecionamento forte pela função $f_{SO}$, que consiste em caracterizar e exemplificar os mecanismos de redirecionamento implementados basicamente a partir do sistema operacional; no **capítulo 6** são enfocados os mecanismos de redirecionamento implementados a partir da forte interação de $f_L$ e $f_{SO}$ e no **capítulo 7**, finalizamos com as mais importantes conclusões retiradas desta dissertação.

## 1.1 Motivação

Esta dissertação teve como motivação principal reunir todas as entidades que possibilitam explorar os recursos provenientes da interligação de dois ou mais computadores em rede sob um paradigma único. Tal paradigma reúne várias áreas tais como linguagens, sistema operacionais e redes de computadores, retratando seu grau de interrelação. Merecido reconhecimento é dado aos trabalhos desenvolvidos no Sistema CONIC, explanado no capítulo 6, cujas idéias influenciaram significativamente esta dissertação.

# Capítulo 2

# Processamento Concorrente e Distribuído

A obtenção de um ganho real de desempenho e funcionalidade, quando se interconecta vários computadores através de uma rede de comunicação, envolve a exploração de paralelismo em um sistema composto por várias unidades de processamento fracamente acopladas. No desejo de uniformizar a nomenclatura usada e enfatizar as particularidades de um sistema multicomputacional frente a outros sistemas que exploram paralelismo e/ou concorrência, iremos rever, neste capítulo, alguns aspectos básicos na discussão de processamento distribuído. Para atingir tal objetivo este capítulo foi divido em três grandes partes, a saber:

**Programação Concorrente** englobando os principais conceitos pertinentes à programação e ao processamento concorrente tais como: o conceito de granularidade, os mecanismos de especificação de execução concorrente (co-rotinas, *fork* e *join*, *cobegin..coend*, declaração de processos e objetos), as primitivas de comunicação e sincronização entre processos baseadas em memória compartilhada (espera-ocupada, semáforos e monitores) e em troca de mensagens (primitivas, conceituação dos canais de comunicação, sincronização e chamada remota de procedimentos) e os mecanismos para expressar e controlar o "não determinismo".

**Processamento Distribuído** englobando principalmente as diferenças entre o processamento distribuído e concorrente, a taxionomia de um processamento distribuído quanto à arquitetura do sistema, as aplicações que exploram distribuição e o impacto de uma rede de comunicação sobre as primitivas de troca de mensagem.

**Redes de Comunicação** englobando a conceituação de redes locais e de longa distância, a explanação das principais funções de uma rede de comunicação através da apresentação do modelo de referência ISO-OSI e a conceituação do termo TCP-IP.

## 2.1 Programação Concorrente

Um programa seqüencial é a especificação de uma seqüência de comandos que irá atuar sob um determinado conjunto de dados. Denominamos processo um programa em execução.

Um programa concorrente é a especificação de um conjunto de comandos seqüenciais que serão executados em paralelo ou em pseudo-paralelismo (simulando uma execução paralela), de tal forma que haja uma cooperação entre as diversas seqüências de execução. O número de instruções associado a um conjunto de comandos seqüenciais desta natureza determina a granularidade da concorrência implementada, que geralmente é menor ou igual a granularidade do sistema. O conceito de granularidade do sistema será explanado nos parágrafos subseqüentes. Como um programa concorrente lida com várias instruções sendo executadas paralelamente, inexiste uma sincronização natural entre estas atividades que possa caracterizar uma ordem determinística de execução do programa. Assim, para que haja cooperação entre as várias atividades concorrentes definidas em um programa, é necessário que haja adequada comunicação e sincronização entre eles.

A quantidade de instruções seqüenciais que será considerada como uma unidade de concorrência determina a granularidade da concorrência implementada. Todo o sistema que explora alguma forma de concorrência possui uma unidade de concorrência que determina quantas instruções seqüenciais são executadas sem possibilidade de serem, naquele nível, fragmentadas. Denominamos sistemas concorrentes de baixa granularidade os sistemas nos quais a unidade de concorrência é a nível de instrução. Tais sistemas são bem representados por processadores vetoriais e máquinas de fluxo de dados. Sistemas concorrentes de média e grossa granularidade são sistemas onde a unidade de paralelismo é um conjunto de vários comandos seqüenciais. O conceito de granularidade é usado freqüentemente para comparações entre sistemas concorrentes. Via de regra, quanto maior a granularidade menor é a quantidade de comunicação e sincronização entre as unidades de concorrência. Sistemas podem possuir mais de uma granularidade, em diferentes níveis. Assim, um conjunto de instruções pode ser divido em unidades de granularidade média a nível de sistema, que por sua vez podem ser fragmentadas em instruções, em um nível mais inferior. Recursivamente, pode-se chegar a um nível de concorrência (paralelismo) de micro-código.

Estamos interessados em analisar a concorrência de um conjunto de instruções implementado como um processo. Desta forma, um programa concorrente irá definir um conjunto de processos seqüenciais que irão cooperar através de mecanismos de comunicação e sincronização entre eles.

A especificação de um processo dá-se por um programa seqüencial. Linguagens concorrentes, que exploram concorrência no nível de processo, conceitualmente nada mais são do que extensões de linguagens seqüenciais que lidam com mecanismos de comunicação e sincronização entre processos. A programação concorrente designa o desenvolvimento de programas para um ambiente onde processos podem ser executados concorrentemente e cooperam uns com os outros. Nesta atividade abstrai-se de várias questões de implementação.

Existem várias formas de especificar concorrência, de prover concorrência e de tratar concorrência. Podemos especificar concorrência através da linguagem de programação, do sistema operacional, de uma linguagem de configuração ou através de outros mecanismos. Podemos prover concorrência através de um conjunto de processadores compartilhando memória, de um conjunto de computadores independentes que se comunicam trocando mensagens sob rede de comunicação ou até mesmo através da implementação de multitarefa no núcleo do sistema operacional. Podemos lidar com vários conceitos de comunicação e sincronização de processos concorrentes através de semáforos, monitores e outros. Com o objetivo de enfocar as principais questões da "programação concorrente" pertinentes ao

escopo desta dissertação, iremos dividir nossa discussão em três importantes assuntos, a saber:

- A especificação de execução concorrente,
- As primitivas de comunicação e sincronização baseadas:
  - em memória compartilhada,
  - em troca de mensagens,
- Maneiras de expressar e controlar "não determinismo".

### 2.1.1 A especificação de execução concorrente

Conceitualmente a especificação de execução concorrente tem o objetivo de determinar seqüências de instruções que poderão ser executados concorrentemente. Iremos explanar cinco mecanismos de especificação de concorrência, que são os seguintes:

- Co-rotinas,
- *Fork* e *Join*,
- *Cobegin ... Coend*,
- Declaração de processo,
- Objetos.

Uma das técnicas para explorar o paralelismo potencial de um sistema é possibilitar uma especificação implícita de concorrência, usada em alguns sistemas verdadeiramente paralelos. Assim, os sistemas que exploram concorrência podem implementar alguma variação ou até mesmo um outro mecanismo dos acima enumerados. Abaixo, iremos conceituar e comentar esses cinco itens, considerados clássicos:

**Co-rotinas:** Co-rotinas são como subrotinas, porém, permitem a transferência do fluxo de controle de uma maneira simétrica ao invés de estritamente hierárquica. A transferência do fluxo de controle de um programa concorrente implementado com co-rotinas é feito por um comando `resume`. A semântica de `resume` é semelhante à semântica de um comando *call*, pois, ambos transferem o fluxo de controle para uma rotina especificada, salvando o estado do processamento. Uma rotina invocada através de um comando *call* retorna à rotina original (que a invocou) através de um comando *return*, entretanto, uma co-rotina retorna à co-rotina original apenas através da execução de um outro comando `resume` explicitando a co-rotina original como parâmetro deste comando. Exemplificando, uma co-rotina de nome `CR1` pode invocar `CR2` através de um comando `resume CR2`, que por sua vez pode invocar `CR3` através de um comando `resume CR3`. A co-rotina `CR3` pode retornar o fluxo de controle para `CR1` através de um comando `resume CR1`.

Quando utilizadas com cuidado, co-rotinas são uma maneira aceitável de organizar programas concorrentes que compartilham um simples processador, isto é, a utilização de

co-rotinas é um método aceitável de fazer escalonamento. De fato, multiprocessamento através de co-rotinas pode ser implementado em um simples processador, mas seu uso não é adequado para a implementação de processos verdadeiramente paralelos, pois sua semântica permite apenas a execução de uma co-rotina a cada instante de tempo. Em essência, co-rotinas implementam processos concorrentes no qual o chaveamento dos processos é completamente especificado, ao invés de deixar isto a critério do sistema de execução.

**Fork e Join:** O comando *fork* [Dennis66] atua como uma chamada a um procedimento. Entretanto, a rotina invocada é executada concorrentemente com a rotina invocadora. A rotina que executou o comando *fork* pode executar um comando *join*, esperando assim o término da execução do processo criado com *fork*.

No exemplo abaixo, o programa P2 irá executar concorrentemente com P1 após a execução de `fork P2`.

```
program P1          program P2;
   ...                 ...
fork P2;               ...
   ...                 ...
join P2;               ...
   ...                end
```

Comandos *fork*'s em programação concorrente são como desvios incondicionais em programação seqüencial. O uso indisciplinado de *fork* pode causar um emaranhado não estruturado de seqüências de execução, tal como o uso indiscriminado de comandos incondicionais pode causar uma emaranhado não estruturado de fluxos de controle [WegSmo83].

O sistema Operacional UNIX [Ritchie74] faz uso extensivo de variantes de *fork* e *join*.

**Cobegin....Coend:** O comando *cobegin...coend* é uma maneira estruturada de denotar execução concorrente de um conjunto de comandos. A execução de:

*cobegin* $S1||S2||...Sn$ *coend*

causará a execução concorrente de *S1, S2 ... Sn*. Cada comando *Si* pode ser inclusive um novo bloco de *cobegin...coend*. A execução de um *cobegin...coend* encerra apenas quando a execução de todos os *Si*'s estiver terminada.

A sintaxe de um comando *cobegin...coend* explicita quais rotinas serão executadas concorrentemente, provendo uma estrutura de controle de uma única entrada e uma única saída. Por outro lado, como *coend* é um *join* obrigatório, o comando termina apenas quando a execução do processo mais demorado terminar.

**Declaração de processo:** Programas grandes são freqüentemente estruturados como uma coleção de rotinas seqüenciais executadas concorrentemente. Embora tais rotinas possam ser declaradas como procedimentos e ativadas por meio de *cobegin* ou *fork*, a estrutura de um programa concorrente será mais clara se a declaração da rotina estabelecer que ela será executada concorrentemente. No exemplo abaixo, os processos são explicitamente declarados durante a programação:

```
program produtor-consumidor
   var ...

   process produtor
      begin
       ...
      end

   process consumidor
      begin
       ...
      end
end
```

No exemplo acima, durante a execução do programa, existirá apenas uma instância de cada processo declarado. Algumas linguagens permitem declarar processos e criar múltiplas instanciações para serem executadas concorrentemente.

**Objetos:** Um objeto é uma entidade que encapsula dados e os procedimentos que manipulam estes dados em uma entidade única. Os dados são informações privadas ao objeto que definem o seu estado. O conjunto de procedimentos que manipulam o estado do objeto define o seu comportamento. A única maneira de um programa externo examinar ou modificar o estado interno de um objeto é fazer uma requisição (invocação) a uma das operações passíveis de acesso público. Este tipo de requisição é denominada envio de mensagem. O conjunto de operações associadas a um objeto cria uma interface bem definida para este objeto, possibilitando que a especificação de objetos seja pública e o estado e a implementação das operações sobre este estado sejam privativas. Conceitualmente, objetos não possuem áreas de dado em comum que possibilitem a interação entre eles através do uso de memória compartilhada.

Na declaração de um objeto inclui-se não apenas as declarações das variáveis que compõem o objeto, mas também os procedimentos que operarão sobre ele. Uma declaração desta natureza é denominada classe. Cada objeto é uma instância de alguma classe.

O mecanismo de herança, em linguagens orientadas a objetos, permite que uma classe seja definida como extensão de outra classe, previamente definida. Algumas implementações de linguagem suportam o conceito de objetos, mas não suportam o mecanismo de herança. Estas linguagens são denominados "baseadas em objetos".

Linguagens orientadas a objetos geralmente utilizam um modelo de objetos passivos. Um objeto é ativado quando recebe uma mensagem de outro objeto. Enquanto o destinatário da mensagem estiver ativo, o remetente está esperando pelo resultado de forma passiva. Após retornar o resultado, o destinatário torna-se passivo outra vez e o remetente continua seu processamento. Em qualquer momento, apenas um objeto no sistema é ativo. Concorrência é obtida estendendo o modelo de objetos seqüenciais, permitindo que dois ou mais objetos fiquem ativos simultaneamente. Algumas metodologias são propostas para que isto se faça de uma maneira consistente, a saber:

1. Permitindo que um objeto que recebeu uma mensagem continue sua execução após ter retornado o resultado.

2. Mandando mensagens para vários objetos de uma só vez.

3. Permitindo que o remetente da mensagem prossiga sua execução em paralelo com o destinatário.

Objetos se mostram como uma importante ferramenta para especificar e declarar processamento concorrente e distribuído. Ressaltamos, por outro lado, que a expressão "objetos" é também usada em um sentido amplo. A semântica do termo deve ser inferida do contexto em questão.

### 2.1.2 Primitivas de comunicação e sincronização

Para uniformizarmos a nomenclatura e delinearmos um escopo para nossa dicussão iremos considerar concorrência no nível de processos. Assim, um programa concorrente possui dois ou mais processos que cooperam para atingir o objetivo do programa como um todo. Dois aspectos importantes desta cooperação são:

- Comunicação: dados são transferidos de um processo para outro;
- Sincronização: envolve a transmissão de informação do estado de um processo ou do sistema.

A questão da sincronização mostra-se intimamente ligada à comunicação e pertinente às pecularíedades de um programa concorrente. Processos são executados em velocidades imprevisíveis. Assim, para que dois processos se comuniquem entre si, um deve executar alguma ação para sinalizar sua intenção ao outro processo, através da alteração de um valor contido em uma variável ou através do envio de uma mensagem. A ordem implícita de "executar uma ação" e "detectar uma ação" impõe uma sincronização entre os dois processos. Mecanismos de sincronização podem atuar de duas maneiras: atrasando a execução de um processo até que uma dada condição seja verdadeira ou assegurando a execução de um bloco de comandos (instruções) como uma operação indivisível. Os mecanismos de sincronização e comunicação entre processos são comumente denonimados IPC[1].

Iremos, nas seções abaixo, abordar questões fundamentais pertinentes à IPC baseada em memória compartilhada e à IPC baseada em troca de mensagens.

#### Primitivas de Comunicação Baseadas em Memória Compartilhada

Descreveremos a seguir, as primitivas clássicas usadas para sincronizar e estabelecer comunicação entre processos baseadas em compartilhamento de memória, que são:

- Espera ocupada,
- Semáforos,
- Monitores.

[1] Em inglês: *Interprocess Communication*

**Espera ocupada:** Primitivas baseadas em "espera ocupada" foi a primeira e a mais natural tentativa de sincronização entre processos. Para sinalizar uma condição, o processo atribui um valor a uma variável acessível a todos os processos que desejam sincronizar-se. Esperando uma determinada condição, o processo repetidamente testa o valor da variável até encontrar o valor desejado. A técnica de "espera ocupada" *(busy-waiting)* tem como principal desvantagem a perda de processamento, quando na espera do valor desejado.

**Semáforos:** Dijkstra foi um dos primeiros a apreciar as dificuldades de se usar o mecanismo de "espera ocupada" e de outras soluções por *hardware* encontradas até então, propondo o mecanismo de semáforos para sincronização de processos [AndSch83].

Um semáforo é uma posição de memória contendo um valor inteiro não negativo sobre o qual duas operações são definidas: $P$ e $V$. Semáforos são implementados sem espera ocupada, fazendo com que os processos que tentem fazer uma operação $P$ sobre um semáforo $s$, quando $s = 0$, fiquem bloqueados suspendendo a sua execução. A operação $V(s)$ desbloqueará um dos eventuais processos bloqueados. Se um processo ao executar $P$ encontrar a posição de memória $s$ um valor diferente de zero, a sua execução não é suspensa. O mecanismo de semáforos possui a filosofia de interrupção, pois a execução de $V$ sinaliza uma nova condição, implicando na ação de desbloqueio de um processo suspenso.

Semáforos, por não serem primitivas estruturadas, são de difícil utilização. No caso da omissão de uma operação $P$ ou $V$ ou da codificação acidental de $P$ em lugar de $V$, pode-se ter efeitos desastrosos, visto que a exclusão mútua pretendida através do uso de semáforos não poderá ser assegurada.

**Monitores:** Ao contrário dos semáforos, monitores provêem uma primitiva estruturada de programação concorrente. Um monitor possibilita o encapsulamento de um conjunto de procedimentos e dados em uma entidadade única, assegurando que todos os procedimentos definidos dentro do monitor são executados sob exclusão mútua.

Um monitor consiste de: um conjunto de procedimentos que são executados sob exclusão mútua; um conjunto de dados persistentes (denominado também de variáveis permamentes) que é privativo ao monitor, porém, público a todos os procedimentos internos ao monitor; uma interface que possibilita rotinas externas solicitarem serviços do monitor através de chamada a procedimentos do monitor que foram declaradas públicas na sua interface; e, eventualmente, um código de inicialização que é executado apenas uma vez, antes que qualquer outro procedimento no monitor seja executado. Algumas implementações de monitor consideram que todos os procedimentos declarados dentro do monitor são passíveis de invocação externa, isto é, são públicos. Os procedimentos internos ao monitor podem ter parâmetros e variáveis locais, às quais são atribuídos novos valores para cada ativação do procedimento.

Uma variedade de construções foram propostas para realizar uma sincronização condicional em monitores. Esta sincronização é baseada nos estados das variáveis permamentes que são retidos entre ativações de um monitor. Iremos descrever a proposta de Hoare [Hoare74] que introduz o conceito de "variáveis condicionais". Uma variável condicional é uma extensão de uma variável permanente para a qual são definidas duas operações: `signal` e `wait`. Considerando $C$ uma variável condicional, a execução de

*C*.`wait`

causará o bloqueio do processo. O processo em questão ficará pendente em uma fila associada a variável *C*. A execução de

*C*.`signal`

tem a seguinte semântica: se nenhum processo estiver bloqueado na condição *C*, o processo que executou esta instrução continua; caso contrário, ele é temporariamente suspenso e um processo bloqueado na condição *C* é reativado. Um processo suspenso devido a uma operação de *signal* continua quando não existir nenhum outro processo executando no monitor.

Como o monitor garante exclusão mútua, apenas um processo que está executando dentro do monitor encontra-se ativo, porém, um ou mais processos podem estar bloqueados nas filas associadas a cada variável condicional. Este modelo de concorrência é denominado quase-concorrência [Wegner90].

### Primitivas de Comunicação Baseadas em Troca de Mensagens

Nas primitivas baseadas em IPC por passagem de mensagens, mensagens são enviadas e recebidas, ao invés de variáveis compartilhadas serem escritas e lidas. A comunicação ocorre porque um processo, após receber uma mensagem, obtém os valores que foram enviados para ele através da mensagem em questão. A sincronização ocorre porque a mensagem pode ser recebida somente após ser enviada, limitando temporalmente a ordem na qual estes dois eventos podem ocorrer.

Um paradigma de interação entre processos comumente implementado a partir de primitivas de comunicação baseadas em troca de mensagens é o relacionamento cliente/servidor. Um processo servidor é um processo que oferece serviços que podem ser invocados por um ou vários processos clientes. Também é possível que um determinado serviço possa ser oferecido por mais de um servidor, principalmente por motivo de eficiência.

As primitivas básicas de uma troca de mensagem são `send` e `receive`. Uma mensagem é enviada quando se executa o comando:

`send` *lista de expressões* `to` *designador do destinatário*

e a mensagem é recebida quando é executado o comando:

`receive` *lista de variáveis* `from` *designador de remetente*

No exemplo acima, as primitivas especificam o destinatário e o remetente, através dos seus respectivos designadores. A *lista de expressões*, depois de devidamente processada, é enviada pela primitiva `send`. Cada expressão presente na lista, agora um valor número, é atribuída às variáveis definidas em *receive*, após a comunicação se efetuar.

O projeto de primitivas baseado em troca de mensagens envolve a escolha da semântica destas primitivas e o seu inter-relacionamento. Como o objetivo de tratar tais questões e

também discutir um importante paradigma baseado em troca de mensagens denominado RPC, iremos dividir nossa discussão em três assuntos, a saber:

- Canais de comunicação,
- Sincronização,
- Chamada Remota de Procedimentos (RPC).

**Canal de comunicação:** Um canal de comunicação é uma abstração de uma rede de comunicação que provê um caminho entre dois processos através do qual dados podem ser trocados. Um canal é definido estabelecendo-se o destinatário e o remetente de uma mensagem. Vários esquemas foram propostos para nomear (identificar) canais de comunicação. O esquema mais simples é designar como remetente e destinatário da mensagem o processo que irá enviar e receber a mensagem, respectivamente. Referimo-nos a esta nomeação como direta. Assim:

`send` *cartão* `to` *executor*

envia a mensagem que pode ser apenas recebida pelo processo *executor*. Similarmente:

`receive` *linha* `from` *executor*

permite receber[2] apenas as mensagens enviadas pelo processo *executor*.

A nomeação direta traz sérias dificuldades para lidar com situações onde processos clientes invocam serviços que são oferecidos por vários servidores. Usando nomeação direta, todo processo servidor deve ter um comando de `receive` para cada processo cliente e todo processo cliente deve ter um comando `send` para cada servidor.

Assim, é necessário um esquema mais sofisticado para definir canais de comunicação. Este esquema é baseado no uso de nomes globais, algumas vezes denominados Caixas Postais[3]. Uma Caixa Postal pode aparecer como designador de destinatário de qualquer processo que execute um comando `send` e como designador de remetente em qualquer processo que contenha comandos `receive`. Assim, mensagens enviadas para uma dada Caixa Postal podem ser recebidas por qualquer processo que executa um `receive` referenciando aquela Caixa Postal. Este esquema é particularmente bem sucedido para programação de interações múltiplos clientes / múltiplos servidores. A implementação de Caixas Postais pode ter um custo alto, especialmente quando existe a necessidade de se propagar um `send` a todos os servidores associados a uma dada Caixa Postal e não existe uma rede de comunicação que faça tal propagação a um custo reduzido.

Um caso especial de Caixa Postal, em que uma Caixa Postal aparece como remetente em comandos `receive` de apenas um processo, é chamada de *port*. *Ports* são simples de implementar, pois todos `receives` que designam um *port* ocorrem no mesmo processo,

[2]No sentido de aceitar
[3]Em inglês: *Mailboxes*

permitindo uma solução direta para o problema múltiplos clientes / único servidor; porém, múltiplos clientes / múltiplos servidores não são facilmente resolvidos com *ports*.

Com os conceitos introduzidos até aqui, podemos concluir que quando o esquema de nomeação direta é usado, a comunicação é um para um, desde que cada processo de comunicação nomeie o outro. Quando a nomeação por *port* é usada, a comunicação pode ser de muitos clientes para apenas um servidor. O esquema mais geral é o da nomeação global, que pode ser de muitos para muitos. Nomeação direta e nomeação por *ports* são casos especiais de nomeação global. Ambos limitam os tipos de interação que podem ser programados diretamente, mas são mais eficientes para implementar.

Designadores de destinatário e remetente podem ser fixados em tempo de compilação (nomeação estática) ou eles podem ser computados em tempo de execução (nomeação dinâmica). Embora largamente usada, a nomeação estática apresenta dois problemas. Primeiro, ela impede quaisquer comunicações entre processos que não foram definidos em tempo de compilação, requerendo uma recompilação a cada alteração na configuração do ambiente, como por exemplo, na inclusão e exclusão de novos processos clientes. Um canal de comunicação também pode ser alocado e nunca ser usado, desperdiçando recursos. Em muitas aplicações, a exemplo de um sistema de arquivos, é desejável alocar dinamicamente canais de comunicaçoẽs para os recursos necessários.

**Sincronização:** Apesar de estar implícito no envio de uma mensagem entre dois processos uma sincronização entre eles (uma mensagem só pode ser recebida após ser enviada), uma questão importante, ligada à passagem de mensagens, se refere à possibilidade do processo "atrasar" (bloquear) quando na emissão de comandos de envio ou aceite de mensagens.

As mensagens, a nível de sincronização entre os processos, podem ser síncronas ou assíncronas.

Nas mensagens síncronas o remetente da mensagem é bloqueado até o processo destinatário aceitar a mensagem (através de um comando *receive*, por exemplo). Assim, os processos destinatário e remetente não apenas trocam dados, mas também se sincronizam. A sincronização seguida de comunicação é denominada *rendezvous*, refletindo o fato de que ambos os processos, através de seus comandos de envio e aceite de mensagens, encontram um momento comum no tempo no qual a comunicação entre os processos se faz presente [WegSmo83].

Na troca de mensagem assíncrona, o remetente não espera o processo destinatário aceitar a mensagem. Conceitualmente, o remetente continua a sua execução imediatamente após mandar a mensagem. A implementação pode suspender o remetente até a mensagem ser copiada para transmissão, mas este atraso não altera a semântica.

No modelo assíncrono existe alguma dificuldade para definir a semântica da comunicação em algumas situações. Se o remetente não esperar o destinatário receber a mensagem, pode ser que existam várias mensagens pendentes enviadas pelo remetente, mas ainda não aceitas pelo destinatário. Se o *buffer* de recepção preservar a ordem de chegada das mensagens, as respostas são enviadas nesta ordem para o remetente. As mensagens pendentes são armazenadas em *buffers* que possivelmente podem ser de tamanho insuficiente para armazenar as mensagens ainda não aceitas. Assim, as mensagens enviadas para o destinatário, na condição do *buffer* de recepção totalmente ocupado, podem ser descartadas ou bloquear o processo remetente até que as mensagens possam ser armazenadas no *buffer*. A primeira

opção faz a passagem de mensagem não confiável, visto que as mensagens são descartadas e a ordem de resposta enviada para o remetente não é preservada. A segunda opção introduz uma sincronização entre os dois processos que pode resultar em *deadlocks*[4] inesperados.

**Chamada Remota de Procedimento** Muitas interações entre processos são bilateriais, isto é, mensagens são enviadas e recebidas por ambos os processos. Como exemplo, podemos citar uma relação cliente/servidor onde o cliente solicita um serviço para o processo servidor e fica esperando o resultado. Tais interações são muitas vezes análogas a uma chamada de procedimento, onde temos: a invocação do procedimento com a passagem dos parâmetros de entrada; a execução do procedimento invocado e a suspensão da rotina invocadora; o retorno dos resultados à rotina que invocou o procedimento (denominado também de passagem dos parâmetros de saída), quando no término da execução do procedimento invocado; e finalmente, o retorno do fluxo de execução à rotina que invocou o procedimento.

Conceitualmente, denomina-se Chamada Remota de Procedimento (RPC) uma relação entre dois processos cujo comportamento simula uma chamada de procedimento, onde os processos não compartilham memória entre si. Como não há compartilhamento de memória entre eles, os processos são potencialmente remotos.

Uma contribuição relevante para a popularização de RPC deve-se à tese de doutoramento de Nelson [Nelson81] e ao artigo de Birell e Nelson [BirNel84]. Nelson propõe uma implementação de RPC baseando-se no conceito de *stubs*. Um *stub* se compara a um representante local de um procedimento remoto. Sob este modelo, uma chamada remota de procedimentos envolve cinco entidades: o usuário, o *stub* do usuário, a comunicação de dados, o *stub* do servidor e o servidor.

A figura abaixo mostra a relação entre estas entidades, onde a comunicação de dados é simbolizada pela ligação entre os dois *stubs*.

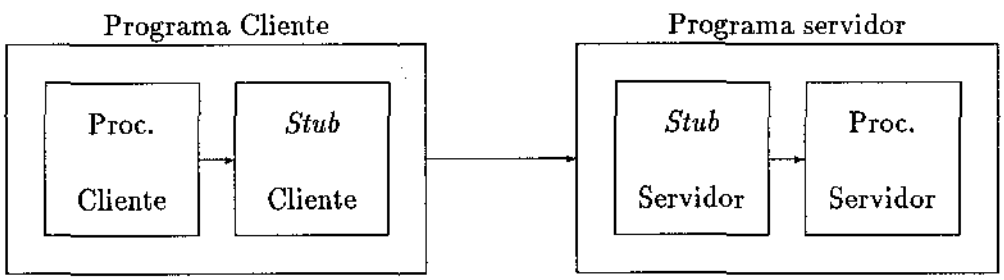



Figura 2.1: Chamada Remota de Procedimento

Quando o processo usuário deseja fazer uma chamada remota, ele faz uma chamada local que envolve o *stub* do usuário correspondente à chamada remota. O *stub* do usuário tem a responsabilidade de colocar a identificação do procedimento a ser chamado e os argumentos a serem passados para a rotina invocada, formatando-os convenientemente para serem repassados para o *RPCruntime*. O *RPCruntime* possui a função de estabelecer uma comunicação de dados confiável entre o *stub* do servidor e o *stub* do usuário. O *stub* do servidor

[4] *Deadlock* é um estado no qual dois ou mais processos estão esperando um evento que nunca irá ocorrer.

recebe os dados enviados pelo *stub* do usuário, através do *RPCruntime*, reformatando-os convenientemente e faz uma chamada local ao procedimento invocado. O processo usuário é suspenso até a chegada do resultado do procedimento invocado. Quando a chamada do procedimento servidor termina, o resultado é passado para o *stub* do servidor, que através do *RPCruntime* transmite para o *stub* do usuário, que por sua vez, recomporá a mensagem enviando para o processo usuário que se desbloqueará.

Assim, processos distintos se comunicam, analogamente a uma chamada de procedimento, sem compartilhamento de memória e por troca de mensagens. As rotinas *stubs* possuem a responsabilidade de manipular as invocações aos procedimentos remotos, transformar os argumentos de entrada e saída em mensagens e remetê-los convenientemente, além de ter algum (ou total) controle das mensagens a serem enviadas. O *RPCruntime* também tem a importante função de manipular o canal de comunicação, que tende a ser mais complexo em um ambiente distribuído, como iremos explanar na próxima seção.

### 2.1.3 Maneiras de expressar e controlar o "não determinismo"

Definimos "não determinismo", neste contexto, como a impossibilidade de predizer o comportamento de um programa, quando em execução. Várias situações pertinentes à computação de processos concorrentes e até mesmo de processos seqüenciais que lidam com entradas que devem ser tratadas em tempo real, apresentam um comportamento não determinístico. Por exemplo, um servidor que oferece serviços de manipulação de arquivos para um sistema desconhece a ordem em que os possíveis clientes irão requerer seus serviços. Programas de tempo real manipulam eventos que ocorrem de modo não determinísticos e que devem ser tratados de acordo restrições temporais. O não cumprimento destas restrições temporais pode implicar na perda de dados de entrada ou no significado dos dados de saída.

Dijkstra [Dijkst75] discutiu o projeto e a implementação de programas que melhor pudessem expressar e controlar o "não determinismo". Foi proposto, então, o conceito de "comando com guarda[5]". Um comando com guarda consiste de uma expressão lógica, seguida por uma série de instruções do programa, denominada "lista com guarda[6]". Uma lista com guarda é passível de execução apenas quando sua "guarda" for `verdadeiro`. Por exemplo, o comando

```
X > Y --> MAX := X
```

só pode ser executado, caso `X>Y`.

Comandos com guarda são usados para formar "construções repetitivas[7]" e "construções alternativas[8]".

Uma construção alternativa é delineada pelos símbolos `if` e `fi` e cada comando com guarda é separado do comando com guarda que o sucede por um símbolo `[]`. A ordem em que cada comando com guarda é escolhida é puramente arbitrária não implicando em nenhuma seqüência ou prioridade entre eles. Abaixo, um exemplo de construção alternativa:

[5] Em inglês: *guarded command*
[6] Em inglês: *guarded list*
[7] Em inglês: *repetitive constructs*
[8] Em inglês: *alternative constructs*

```
if
   X >= Y --> max := x
[]
   Y >= X --> max := y
fi
```

Quando a construção alternativa é executada, todos as guardas são calculadas e só os comandos cuja guarda foi calculado como `verdadeiro` são elegíveis para execução. Apenas um comando com guarda é escolhido arbitrariamente entre os elegíveis para execução. Se nenhuma guarda for calculada como `verdadeiro`, o programa é abortado.

Uma construção repetitiva é análoga à construção alternativa, só que os delimitadores passam de `if` e `fi` para `do` e `od` e as guardas agora são testadas repetidamente, ao invés de serem testadas apenas uma vez como na construção alternativa. Só quando todas as guardas forem `falso` é que a construção repetitiva termina, passando o programa a executar a próxima instrução.

Várias linguagens concorrentes implementam variações de comandos com guarda como CSP [Hoare78]. Os trabalhos de Dijkstra sobre "não determinismo" contribuíram significativamente no campo da certificação[9] de programas concorrentes.

## 2.2 Processamento Distribuído

O processamento distribuído consiste na cooperação de vários processadores autônomos que não compartilham memória, mas cooperam mandando mensagens sobre uma rede de comunicação [BallST89]. Uma rede de comunicação é uma entidade composta por elementos de *hardware* e *software* que tem como função principal possibilitar a transferência de dados sobre um canal de comunicação definido a partir de dois processos que estão sendo executados em processadores distintos.

Para prover uma taxionomia quanto à arquitetura do sistema considerando a interconexão dos processadores através de uma rede, introduziu-se o conceito de sistema fracamente e fortemente acoplado. Um sistema distribuído é denominado fortemente acoplado caso a comunicação entre os processadores é rápida e confiável e os processadores estão fisicamente uns próximos aos outros. De outra forma, denomina-se sistema distribuído fracamente acoplado um sistema onde a comunicação entre os processadores é lenta e não confiável e os processadores estão fisicamente dispersos uns dos outros.

Várias aplicações exploram a distribuição com o objetivo de obter um ganho real de funcionalidade e/ou desempenho. Tais aplicações podem ser resumidas em quatro grandes grupos, a saber:

- Aplicações paralelas de alto desempenho. Diferentes partes de um programa podem ser executados em diversos processadores ao mesmo tempo de tal forma que o programa como um todo pode ser executado mais rapidamente.
- Aplicações tolerantes à falhas. Sistemas distribuídos são mais tolerantes à falhas, pois os processadores são autônomos e a falha de um processador, a priori, não afeta o

[9] Metodologias que possuem o objetivo de garantir (certificar) a correção de um programa

correto funcionamento dos demais. A confiabilidade do sistema pode ser melhorada replicando dados ou funções em vários processadores.

- Aplicações usando especialização funcional. Algumas aplicações são melhor estruturadas como uma coleção de serviços, onde cada serviço pode ter um ou mais processadores dedicados. Servidores de arquivo e servidores de impressora são exemplos de potenciais candidatos à especialização funcional.

- Aplicações inerentemente distribuídas. Algumas aplicações são inerentemente distribuídas, como correio eletrônico.

Um sistema distribuído fracamente acoplado mostra-se adequado para a maioria das aplicações que requerem distribuição. Sistemas distribuídos fortemente acoplado mostram-se ideais para aplicações visando alto desempenho baseado na exploração de paralelismo de baixa granularidade. Um sistema baseado em arquitetura hipercubo MIMD [HwaBri85] [iPSC87] e um sistema multicomputacional[10] são exemplos de sistemas fortemente e fracamente acoplados, respectivamente. Iremos, a partir deste momento, restringir nossa discussão à sistemas multicomputacionais.

Um sistema que contém apenas um processador, por intermédio de um núcleo multitarefa, pode prover um ambiente onde processos cooperem apenas por troca de mensagens, sem nenhum compartilhamento de memória entre eles. Porém, a troca de mensagens entre processos que residem em computadores distintos mostra-se bem mais complexa que uma troca de mensagens em um ambiente multitarefa. Podemos enumerar algumas dificuldades:

- Inexistência de um relógio único e centralizado que sincronize todos os processos e a comunicação entre eles.

- Possibilidade de falha. Uma rede de comunicação, via de regra, é sujeita a falhas. Além da própria rede, um ou vários computadores na rede podem falhar. Em um sistema centralizado, o tratamento de falhas é bem mais simplificado, pois o "estado" de cada processador pode ser monitorado e propagado para os demais processadores com facilidade. Geralmente, em um sistema centralizado, quando o sistema falha assume-se que o cliente, o servidor e o canal de comunicação estão completamente destruídos e nada é feito para reviver um desses. Em uma rede, a detecção de falhas na rede ou no processamento de uma requisição remota se faz através de sondas e/ou políticas de temporização, onde, a priori, não se tem uma idéia precisa de quando aconteceu a falha (não existe um relógio global sincronizando todos os processadores da rede) ou até mesmo se aconteceu a falha (um servidor pode demorar muito para processar uma requisição e isto poderia ser considerado, pelo processo cliente, como uma falha).

- Proteção. Como uma rede de comunicação implica que vários computadores estão interconectados, muitas vezes deseja-se proteger alguns recursos ou a própria mensagem que transita na rede de uso não autorizado. Geralmente, se faz necessário a implementação de políticas de proteção a mensagens, aos próprios processos e recursos que são invocados remotamente.

[10] Conjunto de vários computadores interconectados por uma rede de comunicação.

- Diferença de arquitetura e representação de dados. Em uma rede é possível que haja computadores de arquiteturas diferentes. Não existe um padrão natural das estruturas de dados, de tal forma que, por exemplo, um dado representando um inteiro seja de dois *bytes*, onde o *byte* que está na posição mais baixa da memória é o menos significativo. Para resolver esta questão, foram propostas convenções de representações de dados que independam do sistema envolvido [ASN.1] [SUN87]. Desta forma, o compartilhamento de dados entre sistemas com arquiteturas heterogêneas tornou-se possível, porém o compartilhamento de código continua sendo feito através de compartilhamento de código fonte ou através de geração cruzada de código[11].

- Dificuldade de encaminhar uma determinada mensagem. Em uma rede podem existir vários sistemas computacionais e cada sistema com sua própria sub-rede. Desta forma, quando se envia uma mensagem ela deve ser dirigida para o destino adequado. Muitas vezes, pela complexidade da topologia da rede, encontrar o destinatário de uma mensagem não é uma tarefa trivial. As mensagens devem ser devidamente encaminhadas entre as diversas redes ou caminhos até encontrar o seu destinatário. Por outro lado, é importante atribuir aos recursos da rede, sejam eles dispositivos ou serviços, uma identificação única em toda rede.

## 2.3 Redes de Comunicação

Enumeramos, na seção anterior, o impacto de uma rede de comunicação sobre uma simples troca de mensagens baseada nas primitivas *send* e *receive*. Em sistemas distribuídos fracamente acoplados são comumente usados redes locais e redes de longa distância, denominadas também de LAN[12] e WAN[13]. Em uma rede LAN os processadores estão relativamente próximos uns dos outros na ordem de metros, centenas de metros ou até mesmo poucos quilômetros. Em uma rede WAN, os processadores estão fisicamente distantes, possivelmente em continentes diferentes. Uma rede LAN e WAN diferem principalmente da tecnologia de *hardware* adotada, que se reflete em diferentes taxas de transferência de dados e conseqüentemente no tempo de resposta, porém as diferenças entre ambas serão absorvidas pelo modelo discutido nesta seção. Cada processador, dentro deste escopo, será também um nodo do sistema. No contexto desta dissertação, caso não haja nenhuma referência ao contrário, cada nodo será sinônimo de um computador (processador com seu próprio espaço de memória e seu sistema operacional), a exemplo de uma estação de trabalho.

O ímpeto inicial de interconectar computadores data do início dos anos 60, quando havia a necessidade de interligar terminais remotos e dispositivos de entrada de dados nos computadores centrais. Certamente uma das contribuições mais significativas para o desenvolvimento de redes de computadores foi a rede americana ARPAnet. As experiências obtidas no desenvolvimento da rede ARPAnet tiveram influência decisiva no desenvolvimento do modelo ISO-OSI *(International Standart Organization - Open Systems Interconnection)*, que será abordado a seguir.

[11] Geração de código para execução em um outro processador, de arquitetura diferente.
[12] As iniciais das palavras inglesa *Local Area Network*
[13] As iniciais das palavras inglesa *Wide Area Network*

### 2.3.1 Modelo de referência ISO-OSI

Serviços relevantes são atribuídos a uma rede de comunicação. Iremos abordar o modelo de referência ISO-OSI [Zimmer80], que teve como objetivo ser uma arquitetura para a definição, desenvolvimento e validação de padrões para sistemas de processamento distribuído, com a definição da funcionalidade necessária à comunicação entre processos de aplicação residentes em ambientes (*hardware* e *software*) heterogêneos, a distâncias geográficas variáveis.

O modelo de referência ISO-OSI divide a funcionalidade do modelo em 7 níveis, como representado na figura 2.2, onde a camada 1, mais inferior, é a camada física e a camada 7 é a camada de aplicação.

O princípio básico do modelo de referência ISO-OSI é que cada camada provê serviços necessários para a camada mais alta e assim libera as camadas superiores dos serviços implementados pela inferior. Esta metodologia simplifica o projeto de uma camada particular. Dois padrões são definidos em cada camada: um define a interface dos serviços da camada e o outro define os protocolos dos serviços implementados. Assim, os usuários podem usar cada camada desconhecendo os detalhes de como os protocolos são implementados e até mesmo quais os protocolos são usados para prover um determinado serviço.

Na figura 2.2, está representado o modelo com suas 7 camadas. A pilha da esquerda e a pilha da direita representam 2 nodos que estão se comunicando. As camadas de 7 a 4 (aplicação a transporte) implementam protocolos fim-a-fim, isto é, cada protocolo troca informações apenas com o destinatário e o remetente da mensagem. A pilha pequena, no centro, contendo as camadas inferiores (rede, enlace e físico) estão representando quantos nodos intermediários forem necessários para que a mensagem enviada pelo remetente chegue ao destinatário . Estes protocolos não são fim-a-fim e também um nodo intermediário não precisa, necessariamente, implementar as 3 camadas (pode-se implementar apenas a camada física, por exemplo). O programa do usuário, isto é, o *software* aplicativo, acessa a pilha através da camada 7, a camada de apresentação. Assim, dentro do modelo, o *software* de aplicação de cada nodo está acima da camada 7 em suas respectivas pilhas.

Visão deste modelo:

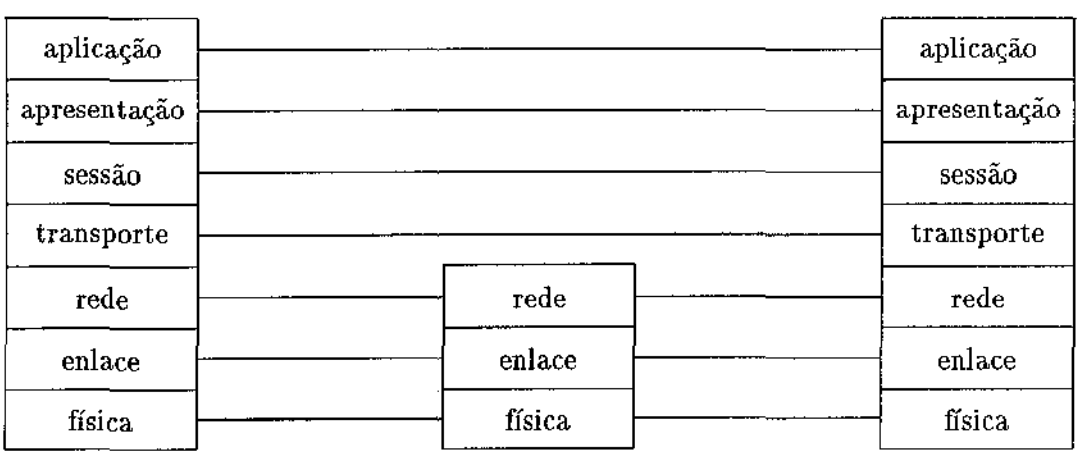



Figura 2.2: Modelo de Referência ISO-OSI

A seguir descreveremos sucintamente a função de cada camada:

**Camada física** Provê codificação, transmissão e decodificação de uma seqüência de bits de uma forma não interpretada de uma máquina para outra.

**Camada de enlace** Provê a troca de dados entre duas entidades da camada de rede. Esta camada detecta e corrige erros que possam ocorrer na camada física.

**Camada de rede** Esta camada gerencia a operação da rede. Mais especificamente, ela é responsável pelo encaminhamento e gerenciamento da troca de mensagens entre duas entidades da camada de transporte. Esta camada é fundamental para o encaminhamento de mensagens para nodos não adjacentes.

**Camada de transporte** Esta quarta camada provê serviços básicos de transferência confiável de dados para entidades da camada de sessão. Esta camada esconde das camadas superiores, e conseqüentemente do usuário da rede, a topologia e as características da rede implementada. A camada de transporte é a primeira camada da pilha ISO-OSI que estabelece uma comunicação fim-a-fim, isto é, a comunicação a nível da camada de transporte se dará apenas entre as entidades destinatária e remetente da camada de transporte.

**Camada de sessão** Esta camada provê serviços necessários às entidades da camada de apresentação, estabelecendo organização e sincronização de seus diálogos. Também está incluso nesta camada, serviços que garantem a segurança na transmissão de dados.

**Camada de apresentação** Esta camada gerencia a representação da informação, da qual as entidades da camada de aplicação fazem uso. Esta camada implementa o mapeamento entre a apresentação da informação usada em diferentes máquinas. As camadas inferiores a esta lidam com a transmissão de dados sem interpretação do seu significado.

**Camada de aplicação** Esta provê as funções de alto nível ou serviços presentes em uma rede de comunicação, tais como correio eletrônico e protocolos para acesso remoto a base de dados.

Assim, os serviços providos pelas camadas no modelo de referência ISO-OSI, permitem estabelecer uma troca de mensagens confiável entre dois nodos e associar a cada nodo um ou vários canais de comunicação, além de oferecer outros serviços que amenizam os problemas de estender o mecanismo de troca de mensagens sobre uma rede de comunicação.

As primitivas básicas de uma troca de mensagem:

`send` *lista de expressões* `to` *designador do destinatário*

`receive` *lista de variáveis* `from` *designador de remetente*

vistos na seção 2.1.2 deste capítulo, estabelecem uma canal de comunicação fim-a-fim entre dois processos. Caso tais primitivas fossem implementadas sobre uma rede de comunicação,

essas seriam pertinentes à camada de transporte, que estabelecem uma remessa de mensagens fim-a-fim entre dois processos.

### 2.3.2 TCP/IP

Como dissemos anteriormente, a rede ARPAnet teve e ainda tem uma influência significativa no desenvolvimento de redes, inclusive no estabelecimento do modelo ISO-OSI. TCP/IP foi originado na rede ARPAnet, mas é usado em todo o mundo abrangendo redes internacionais de instituições de pesquisa, de instituições governamentais e comerciais. O termo genérico "TCP/IP" usualmente significa, ao mesmo tempo, "tudo" e "qualquer coisa" relacionado com os protocolos TCP e IP, incluindo outros protocolos e aplicações [SocoKale91].

Porém, os termos TCP/IP identificam os protocolos TCP *(Transmission Control Protocol)* e IP *(Internet Protocol)* que têm uma funcionalidade comparada à camada 4 (transporte) e camada 3 (rede), respectivamente, do modelo de referência ISO-OSI. O protocolo TCP possibilita um estabelecimento de uma conexão entre duas entidades, provendo um enlace confiável entre elas. O protocolo IP baseia-se em tabelas que possibilitam o encaminhamento de mensagens entre diversos nodos intermediários. Dentro do conjunto de protocolos no qual é definido o protocolo TCP, é também definido o protocolo UDP *(User Datagram Protocol)*. Este protocolo possibilita um envio de dados entre dois nodos sem garantia de entrega e seqüenciamento. Dependendo da aplicação pode-se usar UDP/IP ou TCP/IP.

A explicação detalhada de TCP/IP foge do escopo deste trabalho, porém, uma ótima explanação pode ser encontrada em [SccoKale91].

## 2.4 Conclusão

Analisando este capítulo, conclui-se que a relação entre processamento concorrente, processamento distribuído e redes de comunicação passa pelo nível da abstração presente em cada abordagem. Assim, a abordagem "processamento distribuído" é uma caso particular de "processamento concorrente". Uma rede de comunicação detalha várias particularidades de uma simples troca de mensagem.

Apesar do termo processo ser usado para denotar um programa seqüencial em execução, não raro encontramos vários sinônimos para o conceito de processo, a exemplo de "tarefa". A semântica de um processo em um sistema depende de sua implementação. Um tipo especial de processo que merece nosso destaque é comumente denominado de *thread*. Um *thread* nada mais é do que um processo que compartilha alguns recursos com outros *threads* a exemplo do espaço de memória, com o objetivo de trazer eficiência ao sistema. No capítulo 5 serão abordados alguns sistemas que diferenciam *threads* de processos.

Enfatizamos também o modelo "objetos" como extremamente pertinente a um ambiente distribuído. Uma estratégia que tende a ser bem sucedida para prover distribuição em um sistema é a ação de instanciar objetos em nodos diferentes, que será detalhada nos capítulos subseqüentes.

Em nossa dissertação, abstraímos das particularidades de distribuição de dados e concentraremos na discussão de distribuição de código. Podemos associar a todo dado passível

de distribuição um código, encapsulando-o em um objeto. Porém questões que tratam especificamente de distribuição de dados em um ambiente distribuído fogem do escopo desta dissertação.

# Capítulo 3

# Camada de Redirecionamento

Um sistema multicomputacional possui características que, quando bem exploradas, permitem obter um ganho real em desempenho e funcionalidade não encontrados em outros sistemas, ou mesmo em cada computador separado. Para explorar estas características é necessário integrar, de uma forma adequada, todos os recursos individuais em um recurso único, agora representados pelo sistema. Assim, a interligação de todos os computadores refletirá em ganho caso haja uma efetiva cooperação entre eles. Quanto maior o comprometimento dos elementos que participam desta cooperação - *hardware*, sistema operacional, linguagens e ferramentas de apoio - com o sistema multicomputacional, maior será o ganho de desempenho e funcionalidade do sistema.

Neste capítulo iremos definir uma abstração denominada **camada de redirecionamento** juntamente com um modelo computacional que têm a importante função de possibilitar a identificação, análise e classificação dos diversos elementos que possibilitam a exploração da funcionalidade e desempenho potenciais em um sistema multicomputacional.

## 3.1 Modelo Computacional

Duas entidades são de fundamental importância em um sistema computacional: o sistema operacional, gerenciando e estendendo o *hardware* da máquina, e as linguagens de programação, que transformam abstrações do mundo real em objetos executáveis. O próprio *hardware* é um objeto do mundo real que deve ser modelado em objetos computáveis. Esta visão dual do sistema implica nas seguintes transformações:

$$\text{objeto mundo real} \xrightarrow{f_{ABS}} \text{objeto computável}$$

Na primeira transformação a função $f_{ABS}$ modela um objeto do mundo real para um objeto computável.

$$\text{objeto computável} \xrightarrow{f_L} \text{objeto executável}$$

Na segunda transformação, a função $f_L$ transforma o objeto computável em objetos que possam ser interpretados deterministicamente pela máquina.

$$\text{objeto executável} \xrightarrow{f_{SO}} \begin{array}{c}\text{simulação da abstração do}\\ \text{objeto do mundo real}\\ \text{em um sistema computacional}\end{array}$$

Na terceira etapa, o objeto executável é submetido a $f_{SO}$, tornando-se um objeto em "execução".

Nestas três fases presentes no modelamento de um objeto do mundo real a um sistema computacional, a primeira é inerentemente humana e as demais são providas pelo sistema computacional. A segunda função $f_L$, geralmente um compilador, proporciona um objeto intermediário entre a abstração do objeto e o código a ser submetido a $f_{SO}$. Sob $f_{SO}$, o objeto executável consegue interagir com o sistema computacional, simulando o objeto do mundo real. A função $f_{SO}$ geralmente é implementada pelo Sistema Operacional.

Este modelo representa uma grande parte de sistemas computacionais. Objetos do mundo real são identificados, transformados em algoritmos que são codificados em uma linguagem, produzindo um objeto que será executado sob um sistema operacional.

Visão deste modelo:

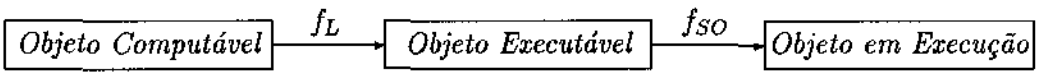



Figura 3.1: Modelo Computacional

## 3.2 Unidade de Processamento

Definimos Unidade de Processamento (UP) como uma entidade capaz de atribuir uma "seqüência de comandos" a um processador virtual. Cada **processador virtual** está associado a um estado que representa o estado de execução da "seqüência de comandos". A definição da UP abstrai da arquitetura da mesma, considerando uma UP como um conjunto de recursos visto como uma entidade única. Os recursos da UP são administrados pela própria UP, através de seu sistema operacional, que está implícito a ela e não visível a outras entidades do sistema.

Pela definição de UP, tanto um sistema multitarefa quanto um monotarefa podem ser tratados como uma UP, diferindo no fato que em um sistema multitarefa pode-se ter vários processadores virtuais enquanto que em um sistema monotarefa existe apenas um processador virtual associado ao processador real. Um sistema multitarefa pode ser tratado como uma UP isolada ou como várias UP's, dependendo do grau de abstração do sistema no qual as UP's estão inseridas. A característica básica de uma UP é a sua capacidade de associar uma "seqüência de comandos" a um processador virtual ou real e executar tais comandos. Uma entidade externa imperativamente atribui uma "seqüência de comandos" a ser execu-

tada em uma UP, porém, a atribuição desta seqüência de comandos à um processador real ou a criação e gerência de processadores virtuais são responsabilidades da UP. Entretanto, nada impede que em um nível de detalhamento maior, todos os processadores virtuais internos a uma UP possam ser transformados em UP isoladas ou que uma UP não possua nenhum processador virtual, apenas o processador real.

Uma UP é uma entidade abstrata, que se molda ao grau de abstração desejado no momento, tendo como característica a possibilidade de execução de uma "seqüência de comandos" em um processador virtual.

## 3.3 Distribuição

Objetos do mundo real passam por várias etapas até se transformarem em "objetos em execução". Um "objeto em execução" pode ter mais de uma "seqüência de comandos" executadas concorrentemente. Cada "seqüência de comandos" ou um "conjunto de seqüências de comandos" poderá ser atribuído a uma UP.

Distribuição, em nosso modelo, é a execução de um objeto computável em mais de uma UP. Se uma UP é associada a um processador físico, dizemos que a distribuição é física. No caso em que uma UP é associada a um processador que simula várias UP's, dizemos que a distribuição é lógica.

Desejamos explorar um sistema que possua mais de uma UP, isto é, um sistema **multi-UP**, distribuindo objetos entre as várias UP's. A distribuição de objetos se dá pelos seguintes mecanismos:

- **A partir de $f_L$:** A função $f_L$ conhece as várias UP's e distribui os objetos entre várias $f_{SO}$'s. Cada $f_{SO}$ é associada a uma UP.
- **A partir de $f_{SO}$:** A função $f_{SO}$ distribui "objetos executáveis" entre várias UP's.
- **A partir de ambos:** Existe uma cooperação entre $f_L$ e $f_{SO}$. A função $f_L$ mapeia e informa as características dos objetos a serem distribuídos por $f_{SO}$.

A decisão de quem e como prover distribuição em um sistema causa sérias conseqüências que implicam no desempenho e funcionalidade de todo um sistema multi-UP.

## 3.4 Camada de Redirecionamento

Na tentativa de identificar mecanismos de redirecionamento, denominaremos "camada de redirecionamento" ao conjunto de todos os objetos (ou entidades) do sistema que provêem mecanismos para que objetos do mundo real sejam mapeados em uma ou mais UP's. Sob a ótica da camada de redirecionamento várias questões são tratadas, objetivando enquadrá-las dentro do modelo computacional proposto neste capítulo e definir sua real contribuição ao redirecionamento do sistema. Tais questões incluem granularidade dos objetos passíveis de distribuição, mecanismos de comunicação interprocessos, servidores de núcleo do sistema, ligação dinâmica e estática, entre várias outras.

O termo "camada de redirecionamento", foi escolhido pelos seguintes motivos:

- O termo "camada" deve-se ao fato que os mecanismos pertinentes à camada de redirecionamento são implementados basicamente pela função $f_{SO}$ e pela função $f_L$, de tal forma que o conjunto destes mecanismos poderia ser um bloco funcional disjunto de ambas funções, localizado conceitualmente entre as funções $f_{SO}$ e $f_L$.

- O termo "redirecionamento" deve-se ao fato de que linguagens como "C" e "Pascal" possibilitam uma nomeação indireta de uma variável ou função através do mecanismo de ponteiros. Ponteiros, nestas linguagens, nada mais são do que variáveis que contêm o endereço de uma variável ou função. Considerando a camada de redirecionamento como uma camada independente, como explanado acima, o termo redirecionamento retrata uma analogia entre a nomeação indireta e a nomeação potencialmente indireta entre um objeto e uma UP na camada de redirecionamento.

A camada de redirecionamento é uma abstração que possibilita identificar e analisar os vários mecanismos de redirecionamento presentes em um sistema e centra-se basicamente nas funções $f_{SO}$ e $f_L$. Utilizaremos as denominações redirecionamento forte por $f_L$ e redirecionamento forte por $f_{SO}$ para identificar a função que implementa os principais mecanismos de redirecionamento do sistema. Um mecanismo pertinente à camada de redirecionamento é denominado fraco, se o mesmo tem uma função secundária na implementação da camada de redirecionamento. Conceitualmente, caso uma função não implemente um redirecionamento forte, poderá ou não implementar um redirecionamento fraco. Tal interação entre $f_{SO}$ e $f_L$ poderá causar a criação de mecanismos de redirecionamento baseados na cooperação entre ambas as funções.

Com o objetivo de padronizar a nomenclatura adotada, renomearemos os três mecanismos de provisão de distribuição, citados acima, como:

- **Redirecionamento forte por $f_L$:** Os mecanismos de redirecionamento do sistema estão baseados principalmente na função $f_L$.

- **Redirecionamento forte por $f_{SO}$:** Os mecanismos de redirecionamento do sistema estão baseados principalmente na função $f_{SO}$.

- **Redirecionamento por $f_L$ e $f_{SO}$:** Os mecanismos de redirecionamento do sistema estão baseados na forte interação entre as funções $f_L$ e $f_{SO}$.

## 3.5 Conclusão

Neste capítulo introduzimos importantes conceitos, tais como o modelo computacional proposto na primeira seção e a camada de redirecionamento, que possibilitaram, conforme veremos nos capítulos posteriores, definir uma taxionomia para sistemas multicomputacionais, relativo à distribuição de código.

Enfatizamos que a camada de redirecionamento é uma entidade abstrata onde a associação da função $f_{SO}$ a um sistema operacional e associação da função $f_L$ a um compilador ou a uma linguagem é meramente conceitual. Várias particularidades de implementação são absorvidas pelo modelo, possibilitando uma visão concisa do sistema em análise.

# Capítulo 4

# Redirecionamento pela função $f_L$

Como visto na explanação de nosso modelo computacional, a função $f_L$ transforma um objeto computável em um objeto executável. O objeto executável é submetido à função $f_{SO}$ a partir da qual é efetivamente executado em uma UP.

Exploraremos, neste capítulo, o modelo de sistema multi-UP onde a função $f_L$ possui a relevante tarefa de prover mecanismos para que objetos executáveis sejam mapeados em uma ou mais UP's, bem como funções ao sistema pertinentes à camada de redirecionamento.

O redirecionamento em um sistema pela função $f_L$ pode ser fraco ou forte. Enfatizaremos neste capítulo sistemas onde os principais mecanismos de redirecionamento se concentram na função $f_L$.

## 4.1 Modelo Computacional

Em nosso modelo, numa primeira etapa, o objeto do mundo real é transformado em um objeto computável através da aplicação da função $f_{ABS}$. Na segunda etapa, o objeto computável é transformado em objeto executável via aplicação da função $f_L$. O redirecionamento pela função $f_L$ caracteriza-se pela geração de objetos executáveis, que podem direta ou indiretamente, invocar e distinguir as várias UP's do sistema, isto é, a função $f_L$ "enxerga" as UP's em um determinado grau de abstração.

Dizemos que existe um "redirecionamento forte via função $f_L$" quando a função $f_L$ é o principal agente de redirecionamento do sistema. Se a função $f_L$ efetivamente contribui para a camada de redirecionamento do sistema, mas não é o principal agente desta camada, então dizemos que a função $f_L$ contribui com um mecanismo de redirecionamento fraco para o sistema.

O redirecionamento por $f_L$ é o método mais intuitivo de se prover distribuição em um sistema. O código necessário para se conseguir que UP's lógica ou fisicamente distintas cooperem, é inserido no código a ser submetido ao sistema operacional e presente nos objetos executáveis. Muitas abstrações foram construídas a partir deste modelo, inclusive suportadas por uma enorme diversidade de linguagens que exploram distribuição.

O redirecionamento pela função $f_L$ caracteriza-se pela geração de objetos executáveis que possuem um determinado grau de comprometimento com a camada de redirecionamento. Se a função $f_L$ é o principal agente da camada de redirecionamento, denominamos este

processo de "redirecionamento forte por $f_L$". Quanto maior o comprometimento de $f_{SO}$ na camada de redirecionamento, maior é o enfraquecimento de $f_L$ nesta camada.
Visão deste modelo:

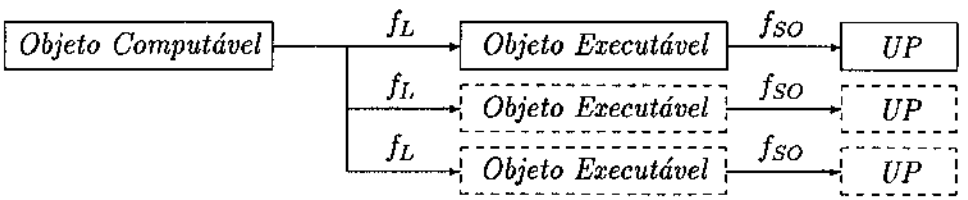



Figura 4.1: Redirecionamento por $f_L$

Na figura acima, a função $f_L$ aloca objetos executáveis a UP's sem que $f_{SO}$ interfira nesta escolha. Este modelo foi utilizado desde os primeiros trabalhos de distribuição em UP's geograficamente distantes e representa o redirecionamento forte por $f_L$.

## 4.2 Implementação

A implementação de $f_L$ se dá principalmente através de compiladores, interpretadores e montadores. Consideraremos interpretadores e montadores como um caso especial de compiladores. Um compilador pode gerar objetos que referenciam outros objetos. Denomina-se "ligação[1]" a resolução destas referências. O processo de ligação é útil, pois permite decompor um objeto grande em vários objetos menores, referenciando cada objeto como um módulo do objeto decomposto. Através do processo de "ligação", a função $f_L$ pode implementar várias funções através de bibliotecas, que poderão ser agregadas explícita ou implicitamente, ao objeto gerado por $f_L$ sem alterar o modelo proposto para a função $f_L$.

A implementação de redirecionamento forte por $f_L$ pode ser caracterizada pela geração de objetos executáveis, que não implementam o redirecionamento pela função $f_{SO}$ - geralmente, o sistema operacional - e sim através de código próprio, embutido ao objeto executável.

## 4.3 Análise de Casos

Nesta seção iremos analisar alguns trabalhos, mostrando a implementação forte da "camada de redirecionamento" pela função $f_L$.

### 4.3.1 Distributed Process

A linguagem Distributed Process (DP), proposta por Brinch Hansen [Brinch78], tem como principal objetivo introduzir conceitos de linguagens de programação em um ambiente onde processos executam concorrentemente comunicando e sincronizando-se através de chamadas

[1] Em inglês: *linking*

de procedimentos e de regiões e comandos com guarda, sem que haja compartilhamento de memória entre os processos.

Regiões e comandos com guarda são mecanismos que habilitam o processo a fazer uma escolha arbitrária entre várias linhas de execução, inspecionando o estado corrente das variáveis associadas a cada possível escolha. Se durante a execução do comando, nenhuma das alternativas for possível (pela inspeção do estado das variáveis), o comando é desconsiderado ou o comando causa a invocação de um procedimento de exceção. Comandos e regiões com guarda são construções que possibilitam controlar e expressar eventos "não determinísticos". Variantes destas construções foram relatadas no capítulo 2 desta dissertação.

Um programa em DP consiste de um número fixo de processos seqüenciais, que são iniciados simultaneamente. Cada processo é associado a um único processador e cada processo pode ter nenhum ou vários procedimentos. O único mecanismo para que os processos se comuniquem é através de uma chamada a algum procedimento externo ao processo. Assim, conceitualmente um programa em DP é executado em um ambiente multiprocessado, onde os processadores não compartilham memória.

Cada processo define seu nome, suas variáveis locais comuns a todo o processo, seus procedimentos e seus comandos de inicialização. Um procedimento, em DP, define seus parâmetros de entrada e saída, suas variáveis locais e os comandos que serão executados, quando o procedimento for invocado. Os procedimentos não são recursivos, porém a linguagem permite que sejam declarados vetores de procedimentos do mesmo tipo.

Um processo começa executando os seus comandos de inicialização até que estes comandos sejam esgotados ou que a execução seja atrasada em uma região com guarda. Qualquer uma destas duas condições possibilita que uma requisição externa a algum procedimento interno ao processo seja aceita. Assim, o processo intercala sua execução entre o processamento dos comandos de inicialização e as requisições externas. Se os comandos de inicialização terminarem, o processo continua a existir e ainda deverá aceitar requisições externas. A execução de um processo é interrompida apenas quando a operação corrente é atrasada em uma região com guarda ou quando o processo faz uma requisição para um processo externo ao procedimento. No primeiro caso, o processo continua sua execução exceto quando todas as suas operações forem atrasadas nas regiões com guarda. No segundo caso, o processo fica esperando até que a requisição externa seja totalmente processada. Nada mais é assumido além destes dois casos, no que diz respeito a ordem de execução dos processos.

Quando um processo faz uma requisição externa, isto é, chama um procedimento externo ao processo, o processo deve explicitar: o procedimento a ser chamado, o processo onde este procedimento deverá ser executado e os parâmetros de entrada e saída. O processo que invocou o procedimento fica esperando até que o procedimento requisitado seja completado. Por outro lado, o processo que irá aceitar esta requisição recebe os parâmetros de entrada e, após o processamento do procedimento especificado, retorna com os resultados que irão atualizar os parâmetros de saída passados por ocasião da requisição do procedimento. Uma requisição externa se faz através de nomeação direta do processo e do procedimento desejado, porém o processo que provê o aceite de uma requisição externa não nomeia (especifica) o processo requisitante.

**Exemplo:** No exemplo abaixo, o processo impressora possui dois procedimentos de nome "requisita" e "libera". Em DP, cada processo está associado a um processador. A seqüência de comandos localizada após os procedimento são os comandos de inicialização. Neste exemplo, a seqüência de inicialização é unitária representada pelo comando da linha 11. A variável `livre` é declarada como do tipo lógica (booleana).

```
1   process impressora; livre:bool
2
3   proc requisita
4    when livre: livre:=false
5   end
6
7   proc libera
8    livre:=true
9   end
10
11  livre:=true
```

Este programa possui a função de permitir que apenas um processo tenha acesso a impressora de cada vez. Assim, um procedimento (externo a este processo) requisitará a impressora executando:

```
impressora.requisita()
```

e ficará bloqueado na linha 4 até o momento em que a variável `livre` passar a assumir o valor `true`. Quando o processo que executou o comando de requisição da impressora com sucesso quiser liberar a impressora, deverá executar:

```
impressora.libera()
```

Qualquer outro processo que requisitar a impressora, quando a mesma já fora anteriormente requisitada e ainda não liberada, ficará bloqueado na linha 4.

É importante notar que em DP toda requisição externa implica em uma chamada remota de procedimento, pois cada processador possui apenas um processo com um ou vários procedimentos. O nome do processo define implicitamente o nome do processador. Apesar de que, neste exemplo, não houve nenhuma passagem de parâmetros, em DP, todos os parâmetros são passados por valor.

### DP no escopo da camada de redirecionamento

Apesar de Distributed Process ser uma proposta de linguagem com o objetivo de introduzir conceitos de programação concorrente e não uma proposta de implementação, é claramente identificável, no ambiente proposto para a linguagem DP, a camada de redirecionamento implementada diretamente sobre a linguagem, que dá ao programador um prévio conhecimento e controle sobre o acesso aos procedimentos locais e remotos.

DP introduz o conceito básico de Chamada Remota de Procedimentos[2], onde processos externos ao processador são invocados, com uma semântica idêntica aos de processos locais. Alguns conceitos indiretos estão agregados à linguaguem como: tipo abstrato de dados, monitores (o conceito de "quase-concorrência" está presente na política de escalonamento das operações, em um processo), nomeação direta (pois na invocação do procedimento, deve-se especificar o processo associado), entre outras.

O redirecionamento encontrado em DP possibilita que os objetos gerados por DP lidem diretamente com processadores distribuídos sobre uma rede de comunicação. Nota-se também que a função $f_{SO}$ é praticamente inexistente, pois todo o controle de alocação de processos e processadores é feito de forma *ad hoc* pelo programador. Tal modelo de redirecionamento, implica no comprometimento da transparência da distribuição de código da camada de redirecionamento.

Pode-se classificar um sistema cujo ambiente é proposto por DP como redirecionamento forte pela função $f_L$.

### 4.3.2 Starmod ou *MOD

A linguagem Starmod [Cook80], é uma linguagem derivada da linguagem Modula [Wirth77] e inspirada em DP. Starmod é baseada no conceito de módulos sob uma rede de comunicação, possibilitando ao programador organizar seus programas para refletir a conectividade (topologia) da rede e a organização do processadores sob esta rede. Na visão da linguagem Starmod, uma rede de computadores pode ser caracterizada como uma coleção arbitrária de processadores com rotas fixas de comunicação para transferência de mensagem.

Starmod usa o conceito de módulo derivado da linguagem Modula como o ponto focal para o desenvolvimento de *software* de rede. Um módulo corresponde a uma abstração de programa e consiste na especificação de sua *interface*, bem como de estruturas de dados, procedimentos, processos e procedimento de inicialização. Dois tipos de módulos são de fundamental importância para Starmod: Módulo de Rede *(Network module)* e Módulo Processador *(Processor module)*. O Módulo de Rede define a conectividade dos processadores e declara tipos, constantes, procedimentos e processos que são globais aos módulos processadores. O Módulo Processador, permite ao programador particionar a computação em uma coleção de processos e procedimentos. A linguagem assume que o processador executa instruções, que um procedimento é uma seqüência de intruções para o processador e que um processo é composto por um ou mais procedimentos compartilhando o estado do vetor que controla e define o processador virtual no qual o processo está executando. Assume-se também que em cada processador exista apenas um módulo processador, apesar de que algumas exceções serem previstas.

Processos no mesmo módulo processador podem-se comunicar usando variáveis compartilhadas ou mensagens, entretanto a comunicação interprocessadores consiste apenas de mensagens. Mensagens podem ter uma estrutura de dados arbitrária ou até mesmo não tê-la, como no caso de interrupções. A linguagem faz a checagem dos tipos das mensagens.

Em Starmod, processos podem ter múltiplas ativações executando em paralelo. Starmod provê um mecanismo de prioridade que tem a função de especificar os relacionamentos entre processos sob o modelo *call/return*. Conforme o valor atribuído à pseudo-variável

[2]Em inglês: *Remote Procedure Call-RPC*

"prioridade", o processo pode ser assíncrono, síncrono, com ou sem mensagem de resposta. Em adição ao mecanismo de *call/return* foi integrado o mecanismo de *ports*. *Port* é uma fila de pedidos de execução de procedimentos ou mensagens para processos, que ao invés de serem atendidos prontamente, passam por um seletor que atende aos pedidos de acordo com uma política não determinística.

**Exemplo:** Este exemplo mostra um pequeno esboço de um programa escrito na linguagem de Starmod, enfatizando o módulo de rede, que define a topologia dos processadores. No caso abaixo, está definida uma rede com topologia estrela, com 4 processadores periféricos e um central. Os processadores comunicam-se por chamada de procedimentos.

```
network module star = (center, pr) , (pr, center);
  const NOPROCESSORS = 4;

  processor module center;
     define   communicator;
     process  communicator(...);
                 .
                 .
        pr[i].communicator(...); (*Chama processador i*)
                 .
                 .
     end      communicator;
     begin   (*inicializacao*)
  end center;


  processor module pr[NOPROCESSORS]
           (*Quatro processadores  perifericos*);
     define communicator;
     import center;
     process communicator(...);
                 .
                 .
          center.communicator(...); (*Chama centro*);
                 .
                 .
          end communicator;
      begin (*inicializacao*)
  end pr;

end star.
```

**Starmod no escopo da camada de redirecionamento**

A linguagem Starmod possibilita o completo direcionamento dos módulos para as respectivas Unidades de Processamento (UP's). A função $f_L$, em Starmod, é responsável pelos caminhos (rotas) de comunicação entre os processadores. Em um programa Starmod, a topologia da rede de comunicações é totalmente discriminada e o programador tem total controle sobre o pedido de chamadas remotas ou locais. A linguagem provê os mecanismos de atribuição de código aos processadores, não delegando esta responsabilidade ao sistema operacional. Ao contrário de DP, Starmod é uma proposta de implementação, porém ambos exemplificam o redirecionamento forte por $f_L$.

### 4.3.3 Emerald

Emerald [Black86] [JulHut88] é uma linguagem de programação concebida com o objetivo de prover suporte de linguagem para a construção de programas distribuídos. Emerald baseia-se no modelo de objetos, no qual, objetos locais, remotos e compartilhados são tratados uniformemente.

Todas as entidades em Emerald são objetos, englobando desde pequenos objetos, a exemplo de objetos designados em linguagens procedimentais do tipo inteiro e booleano, até objetos complexos, a exemplo de diretórios e compiladores. Um objeto pode ser manipulado apenas através da invocação de operações associadas, não sendo permitido que os dados internos ao objeto sejam manipulados diretamente por procedimentos externos a ele. Operações sobre objetos podem ser invocadas remota ou localmente e objetos podem mover-se de um nodo para outro da rede. Cada objeto em Emerald possui quatro componentes:

- Um nome, que identifica o objeto na rede;
- Uma representação de seu estado, que consiste em dados armazenados no objeto. A representação do estado de um objeto definido pelo programador é composta por uma agregação de referências a outros objetos;
- Um conjunto de operações, que define as funções e procedimentos que podem ser executados, no contexto definido pelo objeto;
- Um processo, que é opcional. Um objeto com um processo possui uma existência ativa e executa independentemente de outros objetos, ao contrário de um objeto sem um processo que é um objeto passivo e é ativado apenas por operações efetuadas sobre o mesmo.

Cada objeto possui uma localização que especifica o nodo no qual o objeto está residindo. Objetos, em Emerald, podem ser definidos como imutáveis, podendo ser copiados livremente. Emerald suporta concorrência no âmbito de um mesmo objeto. Desta forma, com um único objeto, várias operações podem ser invocadas simultaneamente sobre o mesmo, gerando vários processos relacionados com o objeto.

Um dos mais importantes objetivos do projeto de Emerald é prover um modelo uniforme de objetos. A semântica de todos os objetos, locais ou distribuídos, pequenos ou

grandes deverá ser única, independentemente da técnica de implementação, tanto para o programador quanto para a aplicação.

Smalltalk [GolRob83] é um bom exemplo de tratamento uniforme de objetos, porém, Smalltalk não é distribuído. Em um ambiente distribuído, diferentes técnicas de implementação podem ser usadas para invocação remota ou local de objetos. Por exemplo, no sistema Argus [Liskov84] existem duas entidades diferentes: *Argus Guardian* que representa a abstração de um nodo e *CLU Cluster* que representa objetos locais dentro dos *Guardians*. Na proposta acima, o programador deve decidir pelo modelo a ser usado, pois ambos modelos são semanticamente distintos. Desta forma, se um objeto é implementado sob um modelo, ele deve ser reescrito para ser implementado sob outro modelo.

Em Emerald, todos os objetos são codificados usando um modelo único e, em tempo de compilação, Emerald escolhe entre diversas formas de implementação, de acordo com o uso do objeto, baseando-se em informações disponíveis durante o processo de compilação. Três diferentes estilos de implementação são usados:

- Objetos Globais - São objetos que podem ser movidos dentro da rede e podem ser acionados por outros objetos não conhecidos em tempo de compilação. Estes objetos são alocados pelo *Emerald kernel* (pequeno núcleo que provê suporte de *run-time* aos objetos globais).

- Objetos Locais - São objetos completamente contidos em outro objeto, isto é, eles não podem ser referenciados fora do objeto que os contém. Tais objetos não podem mover-se independentemente, mas apenas junto com o objeto que os contém. A alocação de um objeto local se faz pelo compilador e não pelo *Emerald kernel*.

- Objetos Diretos - São objetos locais cuja alocação se faz diretamente na representação de dados do objeto que os contém. Objetos diretos são usados principalmente para a construção de tipos, estrutura de tipos primitivos, registros e outras estruturas cuja organização pode ser deduzida em tempo de compilação.

Para os projetistas de Emerald, o paradigma de objetos é ideal para a construção de sistemas distribuídos. Um objeto encapsula os conceitos de processo, procedimento, dado e localização sendo, portanto, para Emerald a unidade básica de distribuição e programação.

Todos os aspectos de distribuição em Emerald são escondidos do programador, mas a localização do objeto é visível. Desta forma, podem ser desenvolvidas aplicações que se beneficiam com a informação sobre a localização do objeto. Por estas razões, Emerald inclui um pequeno número de primitivas de localização, a saber:

- Locate - Localizar um objeto, determinando em qual nodo este objeto reside.

- Fix - Fixar outro objeto em um nodo particular.

- Unfix - Passar o objeto para o estado em que ele é passível de ser movido.

- Move - Mudar um objeto para outra localização.

**Emerald no escopo da camada de redirecionamento**

O projeto Emerald enfoca uma questão importante no modelamento de objetos passíveis de distribuição, que os torna independente de sua implementação. O compilador de Emerald decide a implementação da abstração de um objeto tendo como objetivo a geração de código mais eficiente. O mesmo objeto pode ser recompilado gerando um outro objeto executável, semanticamente idêntico, porém usando outra metodologia de implementação. Este enfoque visa, além do aspecto pragmático da questão, levar ao sistema um modelo onde objetos são tratados de uma maneira uniforme e independente da implementação do mesmo. Outras questões que merecem nossa observação, nesta seção, dizem respeito ao atributo "localidade", associado a um objeto em Emerald e às primitivas de localização, providas pela linguagem.

No nível da camada de redirecionamento, Emerald possui um poderoso mecanismo de redirecionamento, possibilitando a geração de objetos executáveis, onde questões como localidade e mobilidade são devidamente tratadas. Isto é, a linguagem Emerald contribui para o sistema com um importante mecanismo de redirecionamento por $f_L$ gerando objetos executáveis que manipulam e reconhecem as diversas UP's do sistema. Mas, para que o redirecionamento seja forte por $f_L$, é necessário que os mecanismos de redirecionamento sejam implementados pelos objetos executáveis e não pela função $f_{SO}$. Iremos analisar esta questão considerando o núcleo implementado por Emerald.

O núcleo implementado por Emerald deve estar presente em todos os nodos que possuem objetos Emerald em execução. O núcleo pode ser visto como:

- Um objeto Emerald com processo (veja a explanação dos quatro componentes de um objeto em Emerald, na seção anterior);
- Uma coleção de rotinas, que deve estar presente em cada objeto executável gerado pela linguagem e que é dinamicamente ligado aos objetos, em tempo de execução;
- O ambiente de execução dos objetos de Emerald;
- Uma coleção de serviços, não implementada sob o sistema operacional, que possibilita aos objetos possuírem as características desejadas no sistema;
- A união de todos os itens acima.

Dentro do escopo da camada de redirecionamento, podemos ver o núcleo de Emerald como uma coleção de funções que não foram implementadas diretamente em cada objeto gerado por $f_L$, porém devem estar presentes quando um objeto de Emerald estiver sendo executado. Podemos afirmar que Emerald possibilita ao sistema um redirecionamento forte por $f_L$.

## 4.4 Conclusão

Toda linguagem concorrente implementa de alguma forma o redirecionamento por $f_L$. Uma linguagem concorrente garantidamente redireciona várias UP's lógicas dentro de uma UP física.

Na seção "Análise de Casos" deste capítulo, analisamos três linguagens que implementam a camada de redirecionamento gerando objetos executáveis que manipulam UP's sem compartilhamento de memória. Algumas delas enfatizam seu uso em *bare machines*, que são sistemas compostos por estações onde os serviços pertinentes a um sistema operacional são requeridos de uma forma *ad hoc* pelos usuários dos serviços. Casos típicos são: sistemas dedicados para controle de processos e "equipamentos com microprocessador embutido[3]". Outras, a exemplo de Starmod, exige um conhecimento prévio do programador da configuração do sistema (a topologia da rede de comunicação que interliga os processadores, o número de processadores, entre outras, são exemplos de ítens que são pertinentes à configuração de um sistema). Tal esquema, apesar de oferecer uma possibilidade de explorar eficientemente o paralelismo do sistema, oferece uma intrínseca dificuldade na troca de algum elemento do sistema, isto é, na troca da configuração do sistema. Em Starmod, mesmo que o módulo que define a conectividade da rede e os processadores do sistema seja compilado à parte, caso haja uma reconfiguração do sistema, haverá a necessidade de uma nova recompilação de todos os módulos, ou no mínimo, uma nova ligação dos módulos. E, por fim, temos o exemplo de Emerald, que exemplifica um modelo conciso de objetos que engloba também a distribuição de objetos. Emerald possui um núcleo que, além de prover alguns serviços aos objetos do sistema Emerald, tem a importante função de permitir uma grande flexibilidade no gerenciamento da configuração do sistema. O modo de interação de objetos do sistema Emerald com o seu núcleo representa conceitualmente uma ligação dinâmica, onde alguns endereços de subrotinas (operações sobre objetos) são resolvidos em tempo de execução.

Um redirecionamento forte por $f_L$, em outras palavras, significa que os mecanimos implementados sob $f_L$, no sistema, são os principais mecanismos de redirecionamento. Nada implica, contudo, que a função $f_L$ esteja implementada em todo o sistema (em todos os nodos do sistema), mas implica que todo o nodo do sistema que considera a função $f_L$ como o principal mecanismo de redirecionamento do sistema, é implementado o redirecionamento forte por $f_L$. Desta forma, o redirecionamento por $f_L$ pode ser considerado "não global". Os mecanismos de redirecionamento implementados por $f_{SO}$ são mecanismos globais, pois, a princípio, todos os nodos do sistema compartilham a mesma função $f_{SO}$.

A discussão acima, leva-nos também a refletir sobre a tênue separação da camada de redirecionamento implementada sobre $f_L$ e/ou sobre $f_{SO}$. Acreditamos que este trabalho venha a ajudar a enfatizar as diferenças entre ambas. A título de exemplo, podemos refletir sobre a decisão de implementar alguns serviços sobre $f_L$ e/ou sobre $f_{SO}$. Caso se idealize um sistema onde o núcleo do sistema Emerald fosse implementado no núcleo do sistema operacional, teríamos um sistema onde todos os objetos no sistema seriam objetos Emerald, independente da função $f_L$ que está gerando os objetos do sistema. De outra forma, pode-se querer implementar um sistema onde parte dos serviços são implementados pelo objeto, ou por seu *run-time*, e parte dos serviços são implementados pelo sistema operacional. Em outro caso, os serviços podem ser totalmente implementados pelo objetos gerados pela linguagem e/ou por seu *run-time*. A escolha de prover mecanismos de redirecionamento pela função $f_{SO}$ proporciona um ambiente mais homogêneo no tratamento da camada de redirecionamento. Por outro lado, a implementação da camada de redirecionamento por $f_L$ provê uma melhor eficiência no tratamento de objetos, possibilitando gerar objetos que

[3]Em inglês: *embedded microprocessor equipment*

implementam mecanismos de redirecionamento apenas naqueles que realmente vão ser redirecionados. E por fim, sistemas onde o redirecionamento é implementado tanto em $f_L$ como em $f_{SO}$, reúnem qualidades de ambas as abordagens. Nos próximos capítulos trataremos deste assunto.

Enfatizamos que a questão da flexibilidade no tratamento da configuração de um sistema é extremamente importante em um sistema distribuído, pois extensibilidade, reusabilidade e adaptabilidade são requisitos inerentes a estes sistemas.

# Capítulo 5

# Redirecionamento pela função $f_{SO}$

Como visto na explanação de nosso modelo computacional, a função $f_L$ transforma um objeto computável em um objeto executável. O objeto executável é submetido à função $f_{SO}$, a partir da qual é transformado em um objeto que é efetivamente executado em uma UP.

Exploraremos neste capítulo, o modelo de sistema multi-UP onde a função $f_{SO}$ possui a relevante tarefa de prover mecanismos para que objetos executáveis sejam mapeados em uma ou mais UP's, provendo funções ao sistema pertinentes à camada de redirecionamento.

O redirecionamento em um sistema pela função $f_{SO}$ pode ser fraco ou forte. Enfatizaremos neste capítulo sistemas onde os principais mecanismos de redirecionamento se concentram na função $f_{SO}$.

## 5.1 Modelo Computacional

Em nosso modelo, na primeira etapa, o objeto do mundo real é transformado em um objeto computável através da aplicação da função $f_{ABS}$. Na segunda etapa, o objeto computável é transformado em objeto executável via aplicação da função $f_L$. A função $f_{SO}$ caracteriza-se por prover mecanismos para que objetos executáveis possam ser executados pelas UP's.

Dizemos que existe um "redirecionamento forte via função $f_{SO}$" quando a função $f_{SO}$ é o principal agente de redirecionamento do sistema. Se a função $f_{SO}$ efetivamente contribui para a camada de redirecionamento do sistema, mas não é o principal mecanismo de redirecionamento do sistema, então dizemos que este sistema possui um redirecionamento fraco via função $f_{SO}$.

Visão deste modelo:

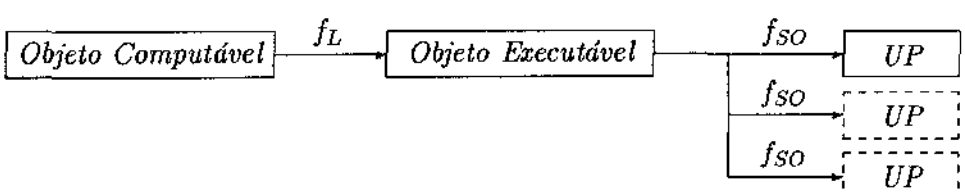



Figura 5.1: Redirecionamento por $f_{SO}$

Na figura 5.1 a função $f_{SO}$ aloca objetos executáveis a UP's sem que $f_L$ interfira nesta escolha. Este modelo é utilizado em vários sistemas operacionais distribuídos e exemplifica o redirecionamento forte por $f_L$.

O redirecionamento por $f_{SO}$ permite uma flexibilidade ao sistema dificilmente encontrada em sistemas que só possuem redirecionamento por $f_L$.

Comparado com o redirecionamento pela função $f_L$, em um redirecionamento por $f_{SO}$ o código necessário para se conseguir que UP's lógica ou fisicamente distintas cooperem é inserido no código do sistema operacional e não está presente em nenhum objeto executável. Os objetos executáveis podem fazer referências às funções implementadas por $f_{SO}$ que perfazem funções da camada de redirecionamento. Neste caso, que não é raro, dizemos que tal sistema possui um forte redirecionamento por $f_{SO}$ e um fraco redirecionamento por $f_L$. Casos especiais onde as funções $f_L$ e $f_{SO}$ possuem um acoplamento forte são discutidas no próximo capítulo.

Muitas abstrações foram construídas a partir deste modelo, presentes em diversos sistemas operacionais distribuídos.

## 5.2 Implementação

Para caracterizarmos as várias implementações da camada de redirecionamento pela função $f_{SO}$, é necessário lembrar que na definição do modelo computacional apresentado com esta camada de redirecionamento fica estabelecido que a função $f_L$ transforma um objeto computável em um objeto executável e a $f_{SO}$ transforma um objeto executável em um objeto em execução. Uma análise mais pormenorizada mostra-nos que o modelo proposto não impõe que um determinado mecanismo deva ser implementado a partir da função $f_{SO}$ ou $f_L$. Exemplificando, um ligador[1] pode ser o principal agente de redirecionamento do sistema e pode ser implementado a partir de $f_{SO}$ ou $f_L$. Geralmente, um ligador estático é implementado a partir de $f_L$ e um ligador dinâmico a partir de $f_{SO}$.

Tipicamente, um sistema que possui redirecionamento forte pela função $f_{SO}$ implementa mecanismos de redirecionamento no núcleo do seu sistema operacional que é referenciado via uma chamada ao sistema ou através de serviços a nível de sistema implementados fora do núcleo, disponíveis em todo o sistema e de conhecimento de todos os nodos do sistema. Um redirecionamento forte pela função $f_{SO}$ libera a função $f_L$ de "enxergar" as várias UP's e de prover objetos executáveis que possuam código para implementar a camada de redirecionamento.

## 5.3 Análise de Casos

Nesta seção iremos analisar alguns trabalhos, mostrando a implementação forte da "camada de redirecionamento" pela função $f_{SO}$.

[1] Em inglês: *linker*

### 5.3.1 Sistema V

O sistema V é um ambiente desenvolvido pela *Stanford University* [Cheriton84] objetivando explorar estações de trabalho interligadas por redes locais de comunicação através de um sistema operacional distribuído. O sistema V baseia-se nos serviços providos pelo núcleo do seu sistema operacional denominado *V kernel*, que implementa uma série de serviços básicos e protocolos de comunicação, de tal forma que o sistema pode ser implementado no nível de processos usuários baseados em um modelo independente de máquina e rede, mas sob os protocolos e serviços básicos do núcleo. Os projetistas do sistema V denominam o *V kernel* como um *software backplane* explorando a analogia entre o *V kernel* e um bom *hardware backplane* que possua *slots* nos quais podem ser colocados módulos e que tenha facilidades de comunicação para que possam interagir.

O *V kernel* possui três grandes componentes: os mecanismos de comunicação entre processos (IPC), o servidor do núcleo e o servidor de dispositivos. Os mecanismos de **IPC** provêem a conexão dos diversos componentes do sistema através da troca de mensagens; o **servidor do núcleo** é uma coleção de rotinas implementadas diretamente no núcleo que provê o gerenciamento de processos e memória como um servidor invocável por IPC e o **servidor de dispositivos** é uma coleção de rotinas implementadas diretamente no núcleo que provê acesso a dispositivos de Entrada/Saída como um servidor invocável por IPC e fazendo uso do *V protocol*. O *V protocol* provê uma interface simples e uniforme, baseada em troca de mensagens, para operações de Entrada/Saída. Tal protocolo permite que sistemas desenvolvidos em outras plataformas como UNIX possuam uma interface para acesso por máquinas que executam o *V kernel*.

**Mecanismos de IPC** As maiores facilidades implementadas no *V kernel* são processos e seu mecanismo de comunicação. O sistema V possui a filosofia de prover IPC como a facilidade básica do sistema e implementar todos os demais serviços em servidores que se comunicam através de IPC. Processos, no sistema V, podem ser criados ou destruídos dinamicamente e são identificados em todo o sistema por uma cadeia de 32 bits, denominada "identificador de processo" ou simplesmente "PID" e comunicam-se através de troca de mensagens. Tanto o destinatário quanto o remetente de uma mensagem são identificados por seu PID. Mensagens podem referenciar opcionalmente áreas de memória denominadas "segmentos" que são manipuladas por operações de cópia inter-processos, possibilitando a referência de um endereço remoto com uma considerável facilidade.

As primitivas *CopyTo* e *CopyFrom* são importantes primitivas baseadas na leitura e escrita de segmentos remotos. A operação *CopyFrom* permite que o processo servidor efetue uma operação de leitura no segmento de memória do processo cliente e, a operação *CopyTo* permite que o processo servidor escreva no segmento de memória do processo cliente. Assim, espaços de memória entre os dois processos (cliente e servidor) podem ser efetivamente compartilhados usando tais primitivas.

Um cenário comum de uma comunicação inter-processos no sistema V é a seguinte: um processo cliente executa uma operação *Send* para um processo servidor que se encontra bloqueado. A chegada da mensagem causa a execução de uma operação *Receive* no processo servidor. Se o cliente passar o direito de acesso a um segmento de memória, o servidor pode copiar dados entre seu espaço de endereços e o do segmento do processo cliente usando

as primitivas *CopyTo* e *CopyFrom*. O processo servidor executa um *Reply* retornando a resposta do serviço desejado ao processo cliente, porém, o servidor pode executar operações *CopyTo* e *CopyFrom* antes do *Reply* ser enviado.

No sistema V, todo envio de mensagem implica na abertura de uma conexão entre os dois processos envolvidos que é fechada quando o *Reply* é enviado. Como explanado acima, as primitivas de cópia inter-processos como *CopyTo* e *CopyFrom* possibilitam uma intensa comunicação antes que o *Reply* seja enviado. Tal metodologia proporciona um eficiente suporte a interfaces procedimentais, inclusive na manipulação de chamadas a rotinas com parâmetros passados por referência, pois o processo servidor pode manipular dados usando operações de cópia inter-processos caso o processo cliente especifique o segmento de memória no qual os dados residem. A correta utilização deste recurso mostra-se eficiente e flexível frente ao processo de serialização[2] dos parâmetros passados para um procedimento remoto usados em várias implementações de RPC. Apesar do eficiente suporte a interfaces procedimentais, o *V kernel* não garante a ordem em que as mensagens serão atendidas, o que poderá trazer alguma dificuldade na implementação do mecanismo de RPC com uma semântica próxima a de uma chamada de procedimento local, devido à ordem aleatória de chegada de mensagens de *Reply* face às sucessivas invocações de procedimentos remotos.

Para assegurar uma interface procedimental, os serviços do núcleo e suas interfaces são encapsulados por rotinas *stubs* que formatam, mandam, recebem e interpretam mensagens, de tal forma que o programador possa desconhecer os mecanismos de troca de mensagens envolvidos nas funções implementadas pelo *V kernel*.

O *V kernel* provê também primitivas para comunicação multiponto, onde uma mensagem é enviada para um grupo de processos. Para que haja comunicação inter-processos, mensagens devem ser trocadas entre processos identificados pelo seu PID. No sistema V existe um grande número de serviços estaticamente alocados, que constituem as funções básicas e são conhecidos por todo o sistema, como o sistema de arquivo. Mas processos dinamicamente alocados podem registrar o seu PID associando um nome que identifica o serviço implementado. O Sistema V possui a facilidade de implementar servidores de nomes de processos, que são manipulados geralmente através de uma comunicação multiponto, resultando assim em uma consulta a um grupo de servidores.

**Servidor do núcleo** é uma coleção de rotinas implementadas diretamente no núcleo que provê gerenciamento de processos e memória. Esta coleção de rotinas comporta-se como um servidor, manipulado através de troca de mensagens, que possui uma interface procedimental, de tal forma que sua implementação preserva a abstração de dados embutida no servidor e a integridade do núcleo.

Um processo, no sistema V, é uma entidade lógica que seqüencialmente executa instruções, na qual é associada prioridade, estado, espaço de endereçamento e de dados. Um processo é criado com o estado inicial *awaiting Reply* possibilitando ao seu criador passar dados iniciais através de primitivas de cópia inter-processos (*Reply* e *CopyTo*). Quando um processo é destruído, todos os processos criados por ele, também são destruídos.

O *V kernel* permite múltiplos processos em um mesmo espaço de endereçamento de memória. Todos os processos que compartilham o mesmo espaço de endereçamento são

[2]Em inglês: *marshalling*

considerados do mesmo *team*. O núcleo provê operações de criar, alocar e liberar memória de um *team*, além de mapear dados dentro do espaço de um *team*. *Teams* provêem uma granularidade mais fina de código e uma eficiente maneira de compartilhar dados entre processos cooperantes.

**Servidor de Dispositivos** é uma coleção de rotinas implementadas diretamente no núcleo provendo acesso e gerenciamento de dispositivos. O servidor interage com os processos clientes através do recebimento e envio de mensagens especificadas em uma interface procedimental, de tal forma que a implementação das rotinas preserva a abstração de dados embutida no servidor e a integridade do núcleo.

O servidor de dispositivos é uma coleção de servidores V que provê serviços de entrada e saída a exemplo de arquivos, *pipes* e redes, fazendo uso de uma maneira uniforme do *V I/O protocol*. A biblioteca de entrada e saída do sistema V provê modelos convencionais de *byte stream* e abertura de arquivos, entre outras, usando o protocolo V e IPC do núcleo.

### Sistema V no escopo da camada de redirecionamento

O sistema V, através de seu *V kernel*, implementa uma das mais importantes características encontrada na camada de redirecionamento, que é a independência de localidade do processo, através dos serviços providos pelo servidor de núcleo e pelos serviços de IPC. Desta forma, objetos executáveis podem ser gerados sem que haja conhecimento prévio de sua localidade e da localidade de outros objetos que possivelmente poderão estabelecer comunicação com ele.

A filosofia de *software backplane* implementada no sistema V, através de seu núcleo, mostra uma flexibilidade adicional à camada de redirecionamento, pois serviços pertinentes à camada, como servidores de nomes e balanceamento dinâmico de carga, podem ser implementados tanto por $f_L$ como por $f_{SO}$ através dos serviços e protocolos básicos oferecidos pelo núcleo.

A função $f_{SO}$, representada pelo *V kernel*, implementa a camada de redirecionamento baseada no tratamento uniforme de objetos locais ou remotos, abstraindo dos serviços que podem ser facilmente agregados pelo usuário à camada de redirecionamento. O redirecionamento forte por $f_{SO}$, no sistema V, proporciona um ambiente onde objetos locais e remotos são manipulados uniformemente, de tal forma que qualidades desejáveis a um sistema distribuído como flexibilidade e extensibilidade são encontrados no sistema. Tais qualidades não são facilmente encontradas em sistemas onde o redirecionamento se dá pela função $f_L$.

## 5.3.2 Sistema Amoeba

O sistema Amoeba é um projeto que foi originalmente desenvolvido na *Vrije Universiteit* em Amsterdam e tinha como principal objetivo a pesquisa das questões ligadas a conexão de vários computadores heterogêneos de uma maneira uniforme [TanRe90]. A idéia básica em Amoeba era a de prover ao usuário a ilusão de um poderoso sistema *multitasking* escondendo o fato de que o sistema foi implementado em uma coleção de máquinas, possivelmente distribuídas em diversos países.

Em 1990, a implementação de Amoeba, encontrava-se em sua versão 4.0, que efetivamente, atingiu marcas consideráveis de desempenho, oferecendo um ambiente no qual a

distribuição se mantém transparente para o usuário. Para atingir tais objetivos, todas as máquinas Amoeba executam o mesmo núcleo que, basicamente, provê processos *multithread*, servidores de comunicação e Entrada/Saída. A idéia básica atrás do núcleo do sistema Amoeba é deixá-lo pequeno, objetivando melhorar a confiabilidade e permitir, o máximo possível, que o sistema operacional seja executado como um processo fora do núcleo, isto é, como um processo usuário.

O sistema Amoeba foi projetado considerando a seguinte arquitetura: **estações de trabalho**, isto é, computadores com rápida resposta interativa usados primariamente como terminais inteligentes para gerenciamento de programas de janelamento; **conjunto de processadores**, de tal forma que os processadores possam ser alocados dinamicamente conforme a necessidade; **servidores especializados** como servidores de arquivos, de dados, de *boot* e de outros servidores que realizam uma função específica e **gateways** que permitem conectar sistemas Amoeba de diferentes países, isolando Amoeba de pecularíedades de protocolo encontradas nas redes de comunicação de longa distância[3].

Amoeba é visto como uma coleção de objetos, onde a cada objeto está associada uma coleção de operações que podem ser efetuadas sobre ele. A cada objeto também está associada uma *capability*, uma espécie de chave de acesso, que permite ao seu possuidor perfazer algumas (não necessariamente todas) operações sobre o objeto. Cada processo usuário possui uma coleção de *capabilities*, que definem o conjunto de objetos que ele pode manipular e o tipo de operações que podem ser efetuadas. As *capabilities* são protegidas criptograficamente do uso indevido usando o mecanismo de *F-box* [MulTan86] e provêem um mecanismo unificado para nomeação, acesso e proteção de objetos. Do ponto de vista do usuário do sistema Amoeba, a função do sistema operacional é proporcionar um ambiente onde objetos possam ser manipulados e criados de forma protegida.

O modelo baseado em objetos, provido pelo sistema Amoeba é visível aos usuários através da implementação de *Remote Procedure Call* (RPC). Associado a cada objeto existe um processo servidor que o gerencia. Quando o processo usuário deseja fazer alguma operação sobre um objeto, o processo envia uma mensagem de requisição para o gerenciador deste objeto. A mensagem contém a *capability* do objeto, a especificação da operação a ser efetuada e eventuais outros parâmetros de que a operação necessite. O usuário (cliente) então bloqueia e quando o servidor retorna a mensagem de resposta, o cliente é desbloqueado.

A combinação da requisição de serviços de cliente para o servidor e a resposta do servidor para o cliente, em Amoeba, constitui uma operação remota. Amoeba baseia-se em três primitivas para fazer operações remotas: *get_request*; *put_reply* e *do_operation*. A primitiva *get_request* é executada pelo processo servidor quando estiver preparado para aceitar requisições do cliente; *put_reply* é executada pelo servidor para retornar a resposta ao cliente do serviço requisitado e a primitiva *do_operation* é executada pelo cliente para invocar uma operação remota.

Em Amoeba, para se fazer uma chamada remota de procedimentos o cliente deve usar as primitivas acima, transmitir a *capability*, o código de operação e os parâmetros do procedimento que devem ser colocados em um *buffer* de requisição. O servidor, por sua vez, precisa conferir a *capability*, extrair o código de operação bem como os parâmetros da requisição e chamar o procedimento correto, colocando o resultado no *buffer* de resposta. Amoeba não

[3]Em inglês: *Wide Area Network*

possui outras facilidades além da serialização dos parâmetros passados para um procedimento. A ação de colocar parâmetros ou resultados em *buffers* de mensagem se faz pela serialização de seus dados.

Para manipular convenientemente as operações de serialização e troca de mensagens, ocultando-as do usuário, Amoeba faz uso de rotinas *stubs*. No entanto, a geração automática dessas rotinas a partir do cabeçalho de um procedimento escrito em uma linguagem procedimental, como C ou Modula, é impossível, pois informações essenciais para a geração dessas rotinas, são perdidas.

Para que a geração de eficientes rotinas *stubs* seja automática, Amoeba provê a linguagem de interface AIL (*Amoeba Interface Language*) na qual a informação extra para a geração de eficientes rotinas *stubs* pode ser especificada. Como AIL trata da serialização dos dados passados como parâmetros, AIL implementa mecanismos que lidam com diferentes representações de dados. Esta característica é importante quando as máquinas que compõem o sistema possuem arquiteturas distintas.

Um processo em Amoeba consiste de um ou mais *threads*, que são executados em paralelo. Todos os *threads* de um processo compartilham o mesmo espaço de endereçamento, mas cada um possui sua própria pilha e contador de programa (PC). Para o programador, *threads* são programas seqüenciais que podem estabelecer comunicação entre eles usando memória compartilhada. *Threads*, opcionalmente, podem sincronizar-se usando semáforos ou exclusão mútua. O objetivo de se ter múltiplos *threads* em um processo é de aumentar o paralelismo em um modelo relativamente simples para o programador. Servidores implementados com múltiplos *threads* geralmente possuem uma performance melhor do que implementados com múltiplos processos de um *thread*.

O núcleo do Amoeba basicamente recebe e manda mensagens, escalona processos e perfaz algumas funções de gerenciamento de memória. As demais funções são realizadas por processos usuários fora do núcleo, inclusive o gerenciamento das *capabilities*. O servidor do sistema de arquivo, o servidor de memória e processos e o servidor de diretórios são importantes servidores do sistema Amoeba que são implementados fora do núcleo, isto é, como processos usuário.

Através do servidor de processos e memória Amoeba cria, bloqueia e destrói processos. Em alguns sistema operacionais não distribuídos, quando um subprocesso é criado, uma cópia exata do processo é feita, a exemplo da primitiva *fork* em UNIX. Amoeba usa uma estratégia diferente para a criação de processos baseada em segmentos e descritores de processos. Um segmento é uma área de memória que pode conter dados ou código. Um descritor de processo é uma estrutura de dados que contém informações sobre o processo não inicializado, imigrado ou depurado. A estratégia usada resume-se em, quando da criação do processo, inicializar o processo com o descritor do processo e em seguida os segmentos de código e dados são modificados através da leitura dos segmentos de código e dados do processo que originou o sub-processo. Com isto, a localização física de todas as máquinas é irrelevante e a migração de processos é facilmente implementada.

Amoeba também lida com redes de longa distância, interligando redes de computadores geograficamente distantes, com o objetivo de prover transparência ao sistema. Amoeba possui o conceito de domínio, onde cada domínio caracteriza-se por uma coleção de redes locais interconectadas. Um *broadcasting* feito por qualquer máquina do domínio é recebido por todas as máquinas pertencentes ao domínio, mas não por máquinas fora do domínio.

**Amoeba no escopo da camada de redirecionamento**

O sistema Amoeba mostra-se semelhante ao sistema V, pois ambos são construídos a partir de um núcleo que provê serviços e protocolos básicos para o sistema. Assim, os demais serviços pertinentes à camada de redirecionamento, não oferecidos pelo núcleo, são agregados ao sistema por servidores externos ao núcleo, provendo um sistema onde extensibilidade, uniformidade de manipulação de objetos remotos ou locais e independência de localidade são facilmente encontrados. O principal agente de redirecionamento do sistema é o núcleo do sistema Amoeba.

Dentro do contexto da camada de redirecionamento, o sistema Amoeba tem como objetivo claro prover um redirecionamento forte em todo o sistema. Além da importante função do núcleo, os mecanismos de proteção dos objetos (através de *capabilities*), a facilidade de lidar com objetos geograficamente distantes e as facilidades para implementar relações cliente-servidor (através das primitivas de comunicação) se mostram importantes mecanismos pertinentes à camada de redirecionamento. O sistema Amoeba faz uso do paradigma de. Objetos, no sistema Amoeba, são as menores unidades de distribuição.

O sistema Amoeba possui os principais mecanismos de redirecionamento implementados na função $f_{SO}$ caracterizando assim um redirecionamento forte por $f_{SO}$. Para que o acesso à estes mecanismos se faça de uma forma mais transparente, o sistema Amoeba oferece uma linguagem de interface (*AIL*). Esta linguagem possibilita que programas procedimentais façam uso de toda a potencialidade e flexibilidade do sistema Amoeba, mais especificamente da sua camada de redirecionamento, ocultando detalhes de IPC e acesso aos serviços do núcleo, pela geração automática de rotinas *stubs*. No nível do modelo computacional proposto juntamente com a camada de redirecionamento, identificamos a linguagem de interface do sistema Amoeba como mais um agente que possibilita que um objeto computável seja transformado em objeto executável, caracterizando assim um redirecionamento por $f_L$. A linguagem *AIL*, apesar da sua importante função de ferramenta de apoio no sistema Amoeba, mostra-se fraca no nível da camada de redirecionamento, pois apenas apóia e não implementa os principais mecanismos de redirecionamento que se encontram no núcleo do sistema.

### 5.3.3 Sistema Mach

Mach é um sistema operacional desenvolvido pela *Carnegie Mellon University* [AccBar86] que tem como objetivo prover novas bases para o desenvolvimento do sistema UNIX, provendo suporte para máquinas mono ou multi-processadas interligadas ou não por uma rede de comunicação.

O sistema Mach baseia-se no *Mach kernel* que provê compatibilidade binária com a versão UNIX 4.3 BSD desenvolvida por *Berkeley* e possui as seguintes facilidades não encontradas em 4.3 BSD: suporte para máquinas multiprocessadas fortemente ou fracamente acopladas, um novo projeto de memória virtual, um novo mecanismo de IPC baseado em *capabilities*. Em adição a estas facilidades, o sistema Mach também provê: um depurador de núcleo (no estilo do popular depurador do UNIX de nome *adb*), um sistema de arquivos remotos e um gerador automático de rotinas *stub* de nome Matchmaker.

Mach retorna ao modelo de UNIX original para prover um sistema extensível que permite aos processos usuários implementarem serviços que poderiam ser apenas integrados ao UNIX

com a adição de código no núcleo do sistema operacional. Para prover esta extensibilidade, Mach baseia-se no modelo de objetos, oferecendo um pequeno conjunto de funções primitivas projetadas para permitir que complexos serviços e recursos sejam referenciados como objetos do sistema. Objetos podem ser arbitrariamente colocados na rede de comunicação sem que o código fonte seja alterado, isto é, objetos possuem independência de localidade. As funções básicas do núcleo do sistema Mach possibilitam que várias configurações do sistema executem em diferentes classes de máquinas, provendo uma interface consistente para todos os recursos. O sistema Mach foi idealizado para que os servidores fossem agregados ao sistema como processos usuários e conseqüentemente não implementados no núcleo. Os mais importantes aspectos do sistema são:

- O acesso aos objetos dos núcleo é efetuado via *capabilities.*
- Todas as abstrações do sistema Mach são extensíveis a máquinas multiprocessadas ou a redes de máquinas mono ou multiprocessadas.
- O mecanismo de comunicação provê suporte para implementação de mecanismos de proteção e transparência da rede.
- Acesso simples à memória virtual sem restrições a alocação, cancelamento de alocação e cópia de memória, permitindo *copy-on-write* e *read-write* compartilhado.
- Todo mecanismo que objetiva explorar paralelismo, implementado em Mach, é projetado para uma ampla variedade de máquinas multiprocessadas, abrangendo arquiteturas com processadores forte ou fracamente acoplados.

Além destes aspectos, Mach prevê compatibilidade com a versão UNIX 4.3 BSD. O núcleo do sistema Mach suporta quatro abstrações básicas:

**Task** é o ambiente onde *threads* podem ser executados e representa a unidade básica de alocação de recursos do sistema. Um processo em UNIX é uma *Task* com um único *thread.*

**Thread** é a unidade básica de utilização da CPU. É equivalente a um "contador de programa" independente operando em uma *task.* Todos os *threads* da *task* compartilham todos os recursos requisitados ao sistema pela *task.*

**Port** é um canal de comunicação e consiste em uma fila de mensagens protegida pelo núcleo, tendo como *Send* e *Receive* suas primitivas fundamentais.

**Message** é uma coleção de dados usados na comunicação entre *threads.*

A princípio, o *Mach kernel* não provê mecanismos que suportem comunicação interprocessos sob uma rede, porém *Mach IPC* permite que os mecanimos de comunicação sejam efetivamente estendidos por um servidor de rede. Um servidor de rede é uma coleção de *tasks* que efetivamente atua como um representante local para *tasks* em nodos remotos, de tal forma que mensagens destinadas para *ports* remotos são enviadas para o servidor de rede. O remetente da mensagem desconhece se o *port* destinatário da mensagem enviada é

local ou remoto. O mecanismo de *port capabilities*, usado para garantir direito de acesso às mensagens, pode ser estendido para utilização em uma rede de computadores. Para garantir a integridade da *capabilities* deve-se codificá-las criptograficamente.

Mach implementa alguns mecanismos de suporte como: um depurador do núcleo no estilo do *adb*; suporte para acesso transparente ao sistema de arquivos remotos e um suporte à linguagem de programação denominado Matchmaker.

Matchmaker [JoRas86] é uma linguagem de especificação de interface provida pelo sistema Mach que compila definições de interface gerando rotinas *stubs* RPC. Matchmaker suporta várias linguagens incluindo C, CommonLisp e Pascal. O principal objetivo de Matchmaker é esconder o mecanismo de troca de mensagens do programador. Desta forma o programador se envolve em receber e enviar mensagens através de uma interface procedimental. Quando se define um objeto em Matchmaker, é declarado um *port* e uma lista de operações de entrada e saída que determinam o tipo das mensagens que serão aceitas ou enviadas por aquele objeto.

A sintaxe de Matchmaker é próxima à sintaxe da linguagem Pascal ou Ada. Constantes, novos tipos de dados e chamadas remotas podem ser declarados. Diferentes tipos de semânticas podem ser especificadas para uma chamada remota de procedimento, em Matchmaker, a saber:

**Remote Procedure** - Matchmaker gera código para o processo cliente enviar uma requisição para o processo servidor e receber mensagens de resposta do servidor. Valores de *timeout* são especificados e a espera pela resposta pode ser assíncrona.

**Message** - Matchmaker gera código para o processo cliente enviar uma messagem de requisição de um serviço sem esperar resposta.

**Server Message** - Matchmaker gera código para o processo servidor enviar uma messagem para o processo cliente.

**Alternate Reply** - Matchmaker gera código para o processo servidor enviar uma resposta para o processo cliente em resposta a um *Remote Procedure* diferente de uma resposta normal, sinalizando a ocorrência de erros durante a execução.

Todas essas variedades de chamadas, exceto *Alternate Reply*, usam o *port* como parâmetro da mensagem enviada. Desta forma, a ligação entre *ports* é feita dinamicamente não usando qualquer método em tempo de compilação ou em tempo de ligação.

Matchmaker manipula uma série de contextos heterogêneos na qual o sistema Mach pode trabalhar com: diversidade de linguagem de programação; diferenças de representação de tipos causados pelas várias linguagens de programação e pelo *hardware*; diversidade de arquiteturas de máquinas multiprocessadas; entre outras.

### Mach dentro do escopo da camada de redirecionamento

Assim como o sistema V e Amoeba, o sistema Mach é construído a partir de um núcleo que provê serviços e protocolos básicos para o sistema. Mach, porém, foi desenvolvido a partir do sistema UNIX de tal forma que Mach, além de prover compatibilidade binária

com UNIX, pretende ser uma nova plataforma de desenvolvimento do sistema UNIX onde extensibilidade, distribuição e paralelismo são explorados.

Os principais mecanismos de redirecionamento estão implementados na função $f_{SO}$, concentrados principalmente no núcleo do sistema. Mach, como Amoeba, faz intenso uso do paradigma objetos, onde objetos, no nível de sistema, são as unidades de distribuição. Agregados ao paradigma objetos encontramos vários elementos, também comuns a Amoeba, que se mostram importantes mecanismos pertinentes à camada de redirecionamento, a exemplo de: mecanismos de proteção dos objetos (através de *capabilities*); a facilidade de lidar com objetos geograficamente distantes e as facilidades para implementar relações cliente-servidor.

Como no sistema Amoeba, Mach possui uma linguagem de interface denominada Matchmaker. As mesmas observações feitas para a linguagem AIL em Amoeba, são válidas para Matchmaker.

Uma importante característica encontrada em Mach, é a possibilidade da ligação dos *ports* em tempo de execução, isto é, os *ports* de um objeto são filas protegidas pelo núcleo que possibilitam troca de mensagens entre objetos em nível de sistema, de tal forma que a conexão entre os *ports* de objetos que desejam interagir pode ser feita dinamicamente, em tempo de execução.

Este mecanismo de ligação dinâmica é de grande importância para a camada de redirecionamento e mostra um bom exemplo de redirecionamento forte por $f_{SO}$ e uma forte interação entre $f_{SO}$ e $f_L$. A função $f_L$ gera objetos que só podem ter seus *ports* totalmante conectados através da função $f_{SO}$, reforçando a camada de redirecionamento do sistema e trazendo flexilidade bem como um mecanismo adicional de distribuição ao sistema.

## 5.4 Conclusão

Neste capítulo enfatizamos mecanismos de redirecionamento forte implementados pela função $f_{SO}$. Este redirecionamento se dá, conceitualmente, pelo sistema operacional, pois o sistema operacional provê mecanismos para que objetos executáveis possam efetivamente ser executados pelas UP's, conforme exposto neste capítulo e no capítulo 3.

Na seção anterior, analisamos três importantes sistemas operacionais, que implementam redirecionamento forte por $f_{SO}$. Apesar de serem sistemas distintos, desenvolvidos em datas, em locais e por equipes diferentes, todos compartilham um grande número de características comuns. A mais notável, explicitada ou não nos respectivos artigos, é a forte influência do sistema UNIX no desenvolvimento destes sistemas, sugerindo que a proposta básica destes sistemas, a exemplo do sistema Mach, é estender o UNIX para aplicações distribuídas, sob um modelo conciso ou, sob outro enfoque, estes sistemas sugerem a idéia de uma proposta de reprojeto do sistema UNIX estendendo-o para aplicações distribuídas.

Uma segunda característica comum, é o projeto destes sistemas baseado em um núcleo pequeno, onde apenas os serviços essenciais ao sistema são implementados no núcleo. Desta forma, os demais serviços do sistema são implementados como externos ao núcleo e desenvolvidos a partir dos serviços oferecidos pelo núcleo. Assim, flexibilidade e extensibilidade são facilmente encontradas, oferecendo um modelo conciso de serviços que visa basicamente o tratamento uniforme de objetos locais ou remotos, independentemente de máquina e dos protocolos de rede.

Uma terceira característica comum, foi o projeto de sistemas baseado no paradigma objetos, pois a idéia de encapsular dados e os procedimentos que atuam sob estes dados, intrínseco ao conceito de objetos, é extremamente útil em sistemas distribuídos, inclusive quando se implementa migração de processos, independência de localidade e outros mecanismos. Há de se ressaltar que o paradigma de objetos é fracamente explorado nos sistemas analisados, inclusive no sistema Amoeba, que afirma ser baseado em objetos. Não se tem, em nenhum deles, o paradigma integralmente aplicado, isto é, o mecanismo de herança, no nível de sistema operacional, não é implementado. Com relação à proteção dos objetos, o mecanismo de *capabilities* mostrou-se simples e vem atender os requisitos de distribuição.

Uma quarta característica comum é a preocupação dos projetistas dos sistemas analisados em prover suporte às linguagens procedimentais de tal forma a ocultar do programador mecanismos pertinentes a um sistema distribuído, a exemplo de Matchmaker e *AIL*, ambas analisadas na seção anterior. Podemos detectar nestes mecanismos de suporte às linguagens procedimentais a influência do trabalho de RPC de Birrel e Nelson [BirNel84] e mecanismos de representação de dados independentes de máquina, a exemplo de XDR [SUN87] e ASN.1 [ASN.1]. Aqui podemos notar que estes sistemas provêem um redirecionamento forte por $f_{SO}$, pois no sistema operacional encontram-se implementados os principais mecanismos de redirecionamento do sistema. Porém, o suporte às linguagens procedimentais providos pelos sistemas analisados demostra ter uma interação fraca com a função $f_L$, atuando como um mecanismo de redirecionamento secundário para dar suporte as funções implementadas por $f_{SO}$. O suporte às linguagens procedimentais vem ocultar do programador a camada de redirecionamento, implementada em $f_{SO}$.

Uma quinta característica comum é a tendência de prover gerenciamento de memória integrado a um gerenciador de processos, de tal forma que sua cooperação permite uma otimização no funcionamento de ambos gerenciadores.

Uma sexta característica comum é a implementação de processos de baixo custo ao sistema, onde todos os processos compartilham uma área de memória comum, a exemplo de *threads* no sistema Amoeba e Mach, e processos de mesmo *team* no sistema V. Porém, tais processos não possuem o mesmo tratamento dos processos convencionais, isto é, um processo de baixo custo não pode ser facilmente trocado por um processo convencional e vice-versa, em tais sistemas. No nível do modelo computacional proposto juntamente com a camada de redirecionamento, notamos que tal procedimento lida diretamente com UP's lógicas. Para que a camada de redirecionamento fosse implementada convenientemente por $f_{SO}$, seria necessário que objetos executáveis fossem mapeados para UP's lógicas e/ou físicas de uma maneira uniforme, o que não acontece com estes sistemas. Este é um bom exemplo de como o modelo da camada de redirecionamento é eficaz, mostrando-nos claramente as limitações dos processos de baixo custo implementados nos sistemas analisados.

Um mecanismo importante, não implementado sob o núcleo, são servidores de nomes que possibilitam identificar processos pelo nome. Dentro deste assunto, enfatizamos a ligação dinâmica, onde referências a objetos são resolvidos em tempo de execução e, não em tempo de compilação. Este assunto será tratado mais detalhadamente no próximo capítulo.

Finalizando, para que haja uma direção forte por $f_{SO}$ é necessário que objetos executáveis sejam mapeados às suas UP's, usando principalmente mecanismos implementados em $f_{SO}$. Conforme visto na seção anterior, a implementação de mecanismos da camada de redirecionamento no núcleo do sistema operacional possibilita a construção de sistemas ope-

racionais flexíveis e extensíveis. Outras funções pertinentes à camada de redirecionamento podem ser implementadas em $f_{SO}$ como servidores fora do núcleo.

Na implementação da camada de redirecionamento em $f_{SO}$ ou em $f_L$, além da diferença conceitual, explanada nas seções anteriores, podemos notar que o redirecionamento por $f_{SO}$ mostra-se como um redirecionamento "global", no sentido que os mecanismos de redirecionamento implementados por esta função estão disponíveis em todos os nodos do sistema e praticamente disponíveis em todos os objetos executáveis do sistema. De outra forma, o redirecionamento por $f_L$ mostra-se mais voltado aos objetos implementados pela função, de tal modo que nem todos os objetos gerados pela função implementam a camada de redirecionamento.

# Capítulo 6

# Redirecionamento pela forte interação entre $f_L$ e $f_{SO}$

Nos capítulos anteriores, enfocamos a camada de redirecionamento implementada por $f_L$ e por $f_{SO}$, individualmente. Este enfoque permitiu caracterizar o redirecionamento forte por ambas funções, porém, pouca relevância foi dada à interação entre ambas.

Neste capítulo enfocaremos alguns mecanismos que pertencem à camada de redirecionamento e exploram a interação entre $f_L$ e $f_{SO}$.

## 6.1 Modelo Computacional

A camada de redirecionamento é um conjunto de entidades que propicia redirecionamento ao sistema. Definimos um redirecionamento como sendo forte por $f_L$, se esta função implementa os principais mecanismos de redirecionamento do sistema e, de forma análoga, definimos o redirecionamento forte por $f_{SO}$. Notamos que o redirecionamento forte por $f_L$ propicia uma camada de redirecionamento quase independente da função $f_{SO}$ e dificilmente se implementa uma camada de redirecionamento forte por $f_{SO}$ sem que haja um comprometimento de $f_L$, isto é, sem que haja um redirecionamento fraco por $f_L$. Tal comportamento é natural, visto que a função $f_L$ gera objetos executáveis que fazem chamadas ao sistema operacional de tal forma que o sistema operacional pode ser considerado como um servidor que oferece vários serviços no nível de sistema para os objetos executáveis.

O modelo computacional de uma camada de redirecionamento que explora a interação entre $f_L$ e $f_{SO}$ em nada diferirá dos modelos apresentados nos capítulos anteriores, visto que haverá um mecanismo de redirecionamento forte por $f_L$ ou $f_{SO}$, no sistema. Entretanto, a forte interação dos mecanismos implementados por $f_L$ e $f_{SO}$ implica na geração de objetos por $f_L$ mais comprometidos com a camada de redirecionamento e, por sua vez, uma função $f_{SO}$ mais dependente das informações embutidas nos objetos gerados por $f_L$. A forte interação dos mecanismos pertinentes à camada de redirecionamento implementados por $f_L$ e $f_{SO}$ permite trazer ao sistema características que lhe conferem uma flexibilidade e funcionalidade dificilmente encontradas em sistemas que não valorizam tal interação.

## 6.2 Implementação

Objetos executáveis gerados por $f_L$, que referenciam agentes de redirecionamento implementados na função $f_{SO}$, lidam com a interação entre ambas funções. Assim, a implementação de agentes de redirecionamento que enfatizam a forte interação entre $f_L$ e $f_{SO}$ se faz pela geração de objetos por $f_L$ que fazem uso, direta ou indiretamente, de importantes mecanismos de redirecionamento implementados em $f_{SO}$. Também a forte interação se dá na geração de objetos que embutem informações que são repassados aos mecanimos de redirecionamento implementados em $f_{SO}$.

## 6.3 Análise de Casos

Nesta seção iremos analisar alguns trabalhos, mostrando mecanismos pertinentes à "camada de redirecionamento" que exploram a interação entre as funções $f_L$ e $f_{SO}$.

### 6.3.1 Sistema Conic

O sistema Conic foi desenvolvido pela *Imperial College of Science & Technology* [KraMag83] [KraMag85] tendo como objetivo prover uma metodologia unificada e integrada para projetar e implementar sistemas distribuídos para controle de processos. Conic consiste em: uma **arquitetura de rede de comunicação** que permite a interconexão de um grande número de computadores; uma **metodologia de software** que suporta um conjunto de componentes reutilizáveis interconectados; de um **sistema operacional distribuído** e um **sistema de comunicação** que provêem suporte para a execução do *software* de aplicação bem como um **gerenciador de sistema** que habilita o sistema a ser modificado e estendido para adaptar-se às necessidades da aplicação.

O sistema Conic possui em sua estrutura de *software* a separação entre **programação de componentes de software**, que são módulos que implementam funções de *software* e/ou interfaciam componentes de *hardware* e a **programação do sistema** que é a construção de sistemas a partir destes componentes de *software*. Denomina-se configuração, em Conic, o estado das interconexões dos componentes de *software* que especificam o sistema. Desta forma, um sistema é construído através de conexões de componentes de *software* que determinam a configuração do sistema.

Conic lida com o conceito de configuração definido como a construção de um sistema de controle a partir de um conjunto de componentes de *software*. Isto envolve a atribuição destes componentes a uma estação de computador e suas interconexões de comunicação. A configuração não é fixa, pois ela deve ser passível de trocas para lidar com falhas e para vir ao encontro das necessidades da aplicação, mesmo quando o programa estiver sendo executado. Assim, a configuração pode ser alterada *on-line*.

Componentes de *software* comunicam-se exclusivamente por troca de mensagens. A interface de um componente de *software* define as mensagens que podem ser transmitidas ou recebidas. O menor componente de *software* que pode ser distribuído ou trocado é denominado **módulo**. Módulo, em Conic, é uma abstração de uma estação, que perfaz alguma função local de monitoramento ou controle, visto que o sistema Conic é orientado para

aplicações de controle de processos. O sistema Conic é configurado a partir de instâncias de módulos.

Para permitir que várias instâncias de módulos similares coexistam num mesmo sistema, instâncias são criadas a partir do tipo do módulo e nomeadas univocamente. A interface que o módulo apresenta para o resto do sistema é dada pelas mensagens que o módulo pode receber ou enviar através de seus *ports* de entrada e saída, respectivamente. A mensagem de uma instância de um módulo para outro é enviada pelo *port* de saída de uma e recebida pelo *port* de entrada de outra instância. A associação entre *port* de entrada e saída é denominada ligação[1]. Um *port* de saída pode ser ligado a um ou mais *ports* de entrada, sendo possível assim uma comunicação multi-destinatária. Muitos *ports* de saída podem ser conectados ao mesmo *port* de entrada, situação típica de um módulo servidor.

Para que haja comunicação, o remetente e o destinatário da mensagem devem concordar na estrutura e representação da mensagem. A garantia é dada checando se os *ports* de entrada e saída envolvidos em uma conexão são do mesmo tipo. Erros podem ocorrer em tempo de execução devido à corrupção de *hardware* na estação ou na linha de transmissão, mas deverão ser detectados por mecanismos do sistema na estação ou na rede.

Conic provê dois conjuntos de primitivas para comunicação inter-tarefas por troca de mensagens. Estas primitivas são usadas para comunicação entre instâncias de módulos, localizados local ou remotamente. O primeiro conjunto de primitivas provê a transação *command-response* e *query-status*. A troca de mensagens sob estas primitivas é tratada como uma transação. O segundo conjunto de primitivas de comunicação provê uma comunicação unidirecional, potencialmente multidestinatária que encontra aplicações em mensagens de alarmes e informações de estado.

O sistema Conic provê um gerenciador de configuração que permite instalar, remover ou modificar componentes de *software* ou *hardware* do sistema, com a função de estender, trocar ou reconfigurar o sistema em caso de falhas. Configurar um sistema de controle distribuído a partir de módulos pré-definidos, em Conic, envolve o carregamento do módulo na estação, a criação de instâncias do tipo do módulo carregado, a conexão (ligação) dos *ports* e a inicialização dos módulos.

Conic conta com dois níveis de linguagem, denominados linguagem de programação e linguagem de configuração. A linguagem de programação é baseada em Pascal com a adição de: primitivas de comunicação inter-processos por troca de mensagens; de cláusulas de declaração de *ports* e *tasks*; de procedimentos de inicialização e ligação do módulo. Os módulos podem ser compilados separadamente desde que eles não contenham referências a outros objetos (exceto declarações de tipo). A linguagem de configuração especifica o conjunto de módulos e suas interconexões. Os módulos são interconectados através da associação de *ports* de entrada e saída. A linguagem de configuração possibilita a troca do sistema simplesmente editando a especificação da configuração e submetendo a nova configuração ao gerenciador do sistema.

O sistema Conic identifica cinco entidades de *hardware* e *software* que provêem o ambiente para instalação, modificação e execução de aplicações Conic, a saber:

1. *Hardware*, constando de estações interligadas por uma rede.

[1] Em inglês: *linking*

2. *Kernel*, um em cada estação, implementando multi-tarefa e IPC via troca de mensagens entre instâncias de módulos localizados na mesma estação, além de prover mecanismos para que o sistema operacional gerencie os recursos da estação e crie dinamicamente instâncias de módulos e ligue seus *ports*.

3. Sistema de comunicação provendo comunicação entre estações.

4. Sistema operacional provendo:

   - Carregamento de módulos dentro da estação,
   - Criação e destruição de instâncias de módulo,
   - Bloqueio ou desbloqueio da execução de uma instância de um módulo.
   - Conexão (ligação) e desconexão[2] de *ports* residentes na estação.
   - Detecção e identificação de erros de programa e falhas de *hardware*.

5. Gerenciador do sistema, provendo facilidades para a instalação e gerenciamento do sistema, trocando a configuração dos componentes de *software*, controlando seu estado e iniciando operações de manutenção e diagnóstico.

**Exemplo:** Para exemplificar as etapas de configuração do sistema em Conic, iremos descrever a instanciação do módulo *pumpcontroller* na estação 1, usando a linguagem de configuração e sua sintaxe. O gerenciador de configuração transladará estas requisições de troca de sistema em comandos para o sistema operacional para que sejam executadas as operações de configuração e reconfiguração.

**Carregamento** A primeira etapa do processo é o carregamento de um módulo na estação desejada. O módulo deverá já estar compilado através da linguagem de programação Conic. A seguinte linha de comando, em Conic, expressa a operação de carregamento:

`LOAD` *pumpcontroller* `AT` *estação1*

**Instanciação** Esta etapa tem a função de prover o ambiente de execução de cada instância especificada. Fazem parte desta etapa funções como: reservar espaço de dados e pilha, executar os comandos de inicialização do módulo, entre outras. Neste exemplo, iremos criar duas instâncias, *pump1* e *pump2* do módulo *pumpcontroller*

`CREATE` *pump1:pumpcontroller* `AT` *estação1*

`CREATE` *pump2:pumpcontroller* `AT` *estação1*

**Conexão** Nesta etapa, o módulo já se encontra instanciado e inicializado, devendo seus *ports* de entrada e saída serem interconectados. A checagem dos tipos dos *ports* é feita para testar compatibilidade. Suponhamos, neste exemplo, que o módulo *pumpcontroller* possua um *port* de nome *cmd* e no sistema exista um módulo *surface* com um *port* de nome *out*. A conexão entre eles, caso o tipo dos seus *ports* sejam compatíveis, se dará pelo seguinte comando:

`LINK` *surface.out* `TO` *pump1.cmd*

[2]Em inglês: *unlink*

**Execução** Nesta estapa, todos as instâncias designadas começam sua execução:

START *pump1, surface*

Os módulos que estão em execução podem ser bloqueados, desconectados e retirados da estação através de comandos enviados para o gerenciador de configuração.

### Sistema Conic no escopo da camada de redirecionamento

O projeto Conic exemplifica o desenvolvimento de um sistema onde foi enfatizado a interação dos diversos elementos que compoẽm um sistema. Esta forte interação possibilitou o surgimento de conceitos como linguagem de "configuração" e troca de configuração em tempo de execução.

O sistema Conic, dentro da abstração da camada de redirecionamento, tem a função $f_{SO}$ implementada principalmente pelo sistema operacional e seu *kernel*. A função $f_L$ é implementada pelas linguagens de configuração e programação. A interação entre ambas é notável. Tanto $f_{SO}$ quanto $f_L$ lidam com módulos e seus *ports*, resultando em uma forte interação entre eles. A $f_L$ através da linguagem de configuração monta um programa a partir da conexão dos *ports* de vários módulos, que é submetido à $f_{SO}$ para execução. Posteriormente, permite-se que sejam quebradas conexões com o programa já em execução, implementando novas conexões inclusive com novos módulos. Isto foi possível graças a um programa ligador incluso no ambiente Conic como parte integrante do gerenciador do sistema, que permite a troca de conexões em tempo de execução.

Vemos em Conic, um ganho real em funcionalidade que advém da interação entre $f_L$ e $f_{SO}$. Por causa desta interação pode-se trocar módulos de um programa, sem a necessidade de reiniciar todo o programa (isto é, todos os módulos do programa). Isto se mostra extremamente útil em sistemas de tempo real e/ou sistemas *on-line*, a exemplo de um sistema centralizado de reservas de passagens aéreas, onde a troca de um módulo do programa sem "tirar do ar" todo o sistema é extremamente desejável. Outros exemplos, principalmente em sistemas de tempo real, são facilmente encontrados na prática.

## 6.3.2 Sun RPC

No capítulo 2 introduzimos os conceitos básicos de RPC. Iremos abordar nesta seção a implementação da *Sun Microsystems Inc.* que consta basicamente de: um protocolo RPC [SUN88]; de um padrão de descrição e codificação de dados independente da arquitetura de máquina, denominado XDR [SUN87]; de um protocolo que mapeia o número do programa remoto e sua versão para um *port* específico da camada de transporte, denominado *portmapper* e de um gerador automático de rotinas denominado *rpcgen*. O objetivo desta seção não é explanar os protocolos em si, mas as conseqüências e características que advém destes.

Este conjunto de ferramentas mostra-se notável, principalmente porque tanto o protocolo RPC quanto o padrão de representação de dados XDR são de domínio público. O protocolo *portmapper* é explanado no apêndice A do documento que especifica o protocolo RPC. O gerador automático de rotinas denominado *rpcgen* é um compilador que esconde do programador detalhes da rede de comunicação, incluindo o padrão XDR. Todos estes mecanismos mostram-se importantes ferramentas na implementação e padronização de RPC.

Iremos ver que os protocolos RPC, *portmapper* e XDR são altamente correlacionados, porém independentes, de tal forma que na implementação do mecanismo de RPC com o protocolo *Sun* pode-se optar por outro mecanismo que não o XDR para padronizar os dados que se comunicam entre diversas máquinas de arquiteturas diferentes, ou até mesmo não tê-lo. Esta independência de funções, característica de programas bem modularizados e do conceito de programação baseada em "objetos", também reflete o desenvolvimento destes protocolos visando uma equivalência funcional direta com camadas de protocolos do modelo de referência ISO-OSI.

**Protocolo RPC** O protocolo RPC consta basicamente da definição das mensagens que são trocadas entre cliente e servidor. Estas mensagens possuem campos que identificam: o tipo da mensagem; a mensagem em si (para detectar retransmissões); o estado da execução da chamada remota; a causa da rejeição da mensagem que invocou o procedimento remoto; o processo cliente e processo servidor através de protocolos de autenticação. Os campos de autenticação são definíveis pelo usuário, isto é, são abertos.

Apesar do protocolo RPC ser baseado no modelo de chamada remota de procedimentos, onde o processo cliente bloqueia até que o resultado retorne do processo servidor, o protocolo não restringe a concorrência do modelo. Assim, é possível, a partir do protocolo RPC, implementar RPC's assíncronas (explorando concorrência entre o processo cliente e o processo servidor, que agora podem ser executados em paralelo), ou um conjunto de processos para implementar o servidor (explorando concorrência entre várias instâncias do processo servidor), ou uma combinação destes.

O protocolo RPC é independente da camada de transporte. Assim, o protocolo não se preocupa como a mensagem foi passada de um processo para o outro. O protocolo lida apenas com a interpretação e especificação das mensagens e não implementa qualquer tipo de mecanismo que ofereça confiabilidade na troca de mensagens. Desta forma, caso o protocolo da camada de transporte não possua troca de mensagens confiável, a aplicação deverá implementar retransmissão e políticas de temporização.

Por causa da independência da camada de transporte, o protocolo não atribui uma semântica específica para a execução de procedimentos remotos. A semântica deve ser inferida a partir da camada de transporte. Por exemplo, considerando os protocolos de comunicação TCP/IP e UDP/IP, uma RPC sendo executada no topo de uma camada de transporte não confiável como UDP/IP cuja aplicação retransmite mensagens RPC após curtos intervalos de tempo, pode inferir, se não recebeu a resposta do processo remoto, que o procedimento foi executado zero ou mais vezes. De outra forma, se o procedimento cliente recebeu uma resposta, ele pode inferir que o procedimento foi executado, no minímo, uma vez.

Um RPC rodando sob uma camada de transporte confiável, como TCP/IP, a aplicação pode inferir, a partir da chegada de uma mensagem de resposta, que o procedimento remoto foi executado exatamente uma vez. Porém, o não recebimento da mensagem de resposta implica que o procedimento invocado pela mensagem enviada pode ter sido executado zero ou uma vez. Ainda deve-se considerar que mesmo quando protocolos orientados à conexão, como TCP/IP, são usados, a aplicação ainda necessita de políticas de temporização e de reconexão para tratar quedas de servidor.

O protocolo RPC possui as seguintes características:

- Uma unívoca especificação do procedimento a ser chamado. Cada procedimento RPC é univocamente definido pelo número do programa, número do procedimento e número da versão. O número do programa especifica um grupo de chamadas remotas correlacionadas, onde cada uma tem seu número de procedimento. Cada programa possui seu número de versão, de tal forma que uma menor troca feita no lado do servidor remoto (adicionando um novo procedimento, por exemplo), gera um programa com o mesmo número do programa anterior, porém com o número de sua versão diferente.

- Autenticação de um cliente para o serviço e vice-versa. O protocolo RPC provê campos suficientes para que um cliente se identifique a um serviço e vice-versa, a cada mensagem de invocação de um serviço ou resposta deste. Vários mecanismos de autenticação são suportados, onde um campo no cabeçalho da mensagem RPC identifica qual mecanismo está sendo usado.

- Um protocolo que possibita que mensagens sejam trocadas entre cliente e servidor, de tal forma que as mensagens esperadas pelo processo cliente corresponda às mensagens enviadas pelo processo servidor e vice-versa.

- A possibilidade de detecção de erros, tais como:

  - Não compatibilidade dos protocolos RPC.
  - Não compatibilidade de versões de protocolo.
  - Erros de protocolo (tal como a especificação incorreta dos parâmetros do procedimento).
  - Razões pelas quais a autenticação remota falha.
  - Razões pelas quais o procedimento remoto não foi executado e/ou chamado.

**Portmapper** Programas clientes necessitam de uma maneira de encontrar programas servidores. Um endereço da camada de transporte que identifica univocamente um canal de comunicação é composto por: um número que identifica a rede; um número que identifica a máquina (nodo) e o número do *port*. Um *port* é um canal de comunicação lógico com a máquina (nodo).

Quando um processo estiver bloqueado esperando em um *port* significa que o processo está esperando receber mensagens da rede de comunicação. A forma pela qual um processo espera em um *port* varia de um sistema operacional para outro, mas todos provêem mecanismos para bloquear um processo até a chegada de uma mensagem no *port*. Assim, mensagens não são enviadas para os processos e sim para os *ports* onde os processos estão esperando por mensagens. O protocolo *portmapper* define um serviço de rede que provê uma metodologia padrão para que os clientes consultem o número do *port* de qualquer serviço remoto disponível no nodo servidor.

A implementação do protocolo *portmapper* consiste em um servidor de números de *ports* cujo acesso é facultado a qualquer usuário da rede. Cada nodo possui apenas uma instância de *portmapper* no momento em que o nodo é inicializado. É importante que seja atribuído a todas as instâncias dos serviços *portmapper* um endereço único e bem conhecido por todos os nodos da rede. Todos os serviços de rede disponíveis no nodo devem ser registrados

em seu *portmapper*, isto é, todos os *ports* dos programas passíveis de serem chamados por clientes remotos devem ser registrados em seus respectivos *portmappers*.

Para encontrar o *port* de um programa remoto, o processo cliente manda uma mensagem RPC para o *portmapper* do nodo servidor. Caso o programa remoto esteja registrado no *portmapper*, é retornado, em uma mensagem RPC de resposta, o número do *port* do programa remoto. O cliente de posse do número do *port* pode enviar mensagens diretamente para o *port* desejado, sem consultar o *portmapper*.

**XDR** XDR é um padrão para descrição e codificação de dados. Isto é útil para transferir dados entre computadores de arquiteturas diferentes.

O padrão XDR define uma linguagem para descrever os formatos dos dados. Esta linguagem é usada apenas para descrever dados e não é uma linguagem de programação. Assim, a linguagem permite descrever formatos complexos de dados de uma maneira concisa. A linguagem XDR é similar a linguagem C.

O padrão XDR assume que *bytes* (ou octectos) são portáteis, onde um *byte* é definido como 8 *bits* de dados. A representação de todos os itens requer múltiplos de 4 *bytes* (ou 32 bits) de dados, numerados de 0 a 'n-1'. Os *bytes* são lidos ou escritos de tal forma que o *byte* 'm' sempre precede o *byte* 'm+1'. Se 'n' bytes são necessários para representar um dado e 'n' não é múltiplo de 4, os *bytes* restantes são preenchidos com bytes de valor zero.

**Exemplo:** A linguagem XDR define alguns tipos e estruturas de dados elementares que possibitam a descrição de qualquer estrutura de dados, por mais complexa que a mesma seja. Abaixo, iremos exemplificar o uso da linguagem XDR. A descrição completa do protocolo juntamente com os tipos de dados XDR podem ser encontrados em [SUN87].

Este exemplo é uma pequena descrição de dados usando XDR para definir os dados presentes em um programa que teria a função de transferir um arquivo entre máquinas.

```
/*Definicao das constantes              Enumeracao
  ------------------------              ----------
 "const" e' uma palavra reservada       "enum" e' uma palavra reservada
 que define um nome simbolico para      que descreve um subconjunto de
 uma constante. Uma constante nao       inteiros (com sinal)
 declara nenhum dado.
*/

const MAXUSERNAME   = 64;    /* tamanho maximo do nome  do usuario */
const MAXFILELENGTH = 13170; /* tamanho maximo do arquivo    */
const MAXNAMELEN    = 255;   /* tamanho maximo do nome do arquivo */

/* Tipos de arquivo */

enum file_kind
    TEXT  =0,      /* Arquivo ASCII       */
    DATA  =1,      /* Arquivo de dados    */
```

```
    EXEC  =2        /* Arquivo executavel  */
};

/*Uniao                                    Cadeia de Caracteres
  ------                                   --------------------
\'union' e' uma palavra reservada          Em XDR uma "string" define
que descreve uma uniao composto            uma cadeia de 'n' caracteres
por um descriminante, seguido por          ASCII, onde os primeiros
um conjunto pre-arranjado de               quatro "bytes" e' codificado
tipos de acordo com o valor do             o tamanho ('n') da cadeia */
descriminante. */

/* Estrutura do arquivo, a partir do tipo do arquivo */
union filetype switch (file_kind kind) {
    case TEXT:                              /* "kind" do tipo */
     void;                                  /* "file_kind" e' */
    case DATA:                              /* descriminante  */
     string creator<MAXNAMELEN>;            /* deste "union   */
    case EXEC:
     string interpretor<MAXNAMELEN>;
};

/*Cadeia de dados opacos
  ----------------------
  Em XDR um tipo "opaque" define uma cadeia de 'n' dados sem
  interpretacao, onde 'n' e' uma constante que especifica
  o numero de bytes necessarios para conter o dado opaco. */

/* Um arquivo completo */
struct file {
    string filename<MAXNAMELEN>;  /*Nome do arquivo*/
    filetype type;                /*informacao  sobre o tipo do arq*/
    string owner<MAXUSERNAME>;    /*Nome do criador do arquivo*/
    opaque data<MAXFILELEN>;      /*dados do arquivo*/
}
```

**RPCGEN** *Rpcgen* é um compilador que tem como objetivo auxiliar o programador a escrever programas RPC, diminuindo os detalhes manipulados na programação. Uma das maiores dificuldades quando se programa RPC é escrever rotinas XDR que convertam os argumentos que são passados para o programa cliente ou o resultado para o formato de rede e vice-versa. O programa *rpcgen* foi desenvolvido pela *Sun Microsystems*, gerando programas que invocam principalmente funções contidas em bibliotecas que implementam os protocolos XDR e RPC. Estas bibliotecas somadas com o programa *rpcgen* reduzem enormemente a complexidade no desenvolvimento de programas que fazem uso de uma rede de comunicação.

*Rpcgen* aceita definições de programas remotos escritos na linguagem RPC. A linguagem RPC é uma extensão da linguagem XDR, com a inclusão dos tipos `program` e `version`. A completa descrição da linguagem XDR e RPC podem ser encontradas em [SUN87] e [SUN88], respectivamente. Entretanto, a linguagem XDR é parecida com a linguagem C.

Genericamente, ao fazer uso do compilador *rpcgen* para construir uma aplicação do tipo cliente-servidor, onde o servidor é um programa potencialmente remoto, devemos:

1. Escrever uma especificação dos serviços remotos na linguagem RPC. A especificação consta em declarar os procedimentos, especificando os argumentos de entrada e saída (retorno). Além disto são atribuídos números para cada procedimento, para o programa e para a versão, conforme convenção especificada no protocolo. Observando a figura 6.1, este arquivo é o `date.x`

2. Compilá-lo usando o compilador *rpcgen*, que gerará cabeçalhos contendo o número do programa, o número da versão do programa e os números dos procedimentos que compoẽm este programa. Este cabeçalho é um arquivo com sufixo `.h`, padrão da linguagem C, que deverá ser incluído nos módulos do cliente e do servidor, através de uma diretiva `#include`. Além disto, *rpcgen* gerará dois *skeletons* na linguagem C, um para ser ligado ao programa servidor e outro para ser ligado ao programa cliente. Um *skeleton* é basicamente rotinas *stubs*, como definido no capítulo 2. Observando a figura 6.1 o cabeçalho esta denominado como `date.h` Os *skeletons* do servidor e do cliente são `date_svc.c` e `date_clnt.c` respectivamente.

3. Escrever as rotinas cliente e servidor considerando que ambas não residirão em um mesma máquina, isto é, que o processo cliente invocará o procedimento remoto e um procedimento servidor será invocado por um cliente remoto. Basicamente, a rotina servidora possui duas pequenas restrições, a saber: a primeira requer que os parâmetros de entrada e saída sejam passados como endereços e a segunda requer que as variáveis que receberão os parâmetros de entrada e de saída sejam declarados como variáveis estáticas[3]. Os endereços passados como parâmetros de entrada e saída são referenciados apenas pelo *skeleton* do servidor. No lado do programa servidor deverão ser usadas funções contidas na biblioteca rpc, que define, entre outras coisas, o protocolo da camada de transporte usado para estabelecer comunicação com o programa servidor. Os procedimentos do programa servidor e do programa cliente estão esquematizadas na figura 6.1 como `rdate.c` e `date_proc.c` respectivamente.

4. Liga-se o *skeleton* do servidor gerado por *rpcgen* com o programa servidor gerando, assim, o programa servidor com seus vários procedimentos. O mesmo procedimento se faz com o programa cliente. O programa servidor e o programa cliente totalmente compilados e ligados estão esquematizadas na figura 6.1 como `date_svc` e `rdata` respectivamente.

5. Coloca-se em execução o programa servidor, na máquina ou máquinas em que se deseja que o serviço esteja disponível.

[3] Variáveis estáticas, no sentido usado na linguagem C

6. Ivoca-se o procedimento remoto por um cliente potencialmente, mas não necessariamente, remoto. A invocação deve identificar a máquina que contém o programa servidor em execução.

Apesar de *rpcgen* gerar módulos na linguagem C, nada impede que o programador escreva procedimentos em outras linguagens, desde que seja observada em todos os módulos (programas) a mesma convenção de chamada de procedimentos.

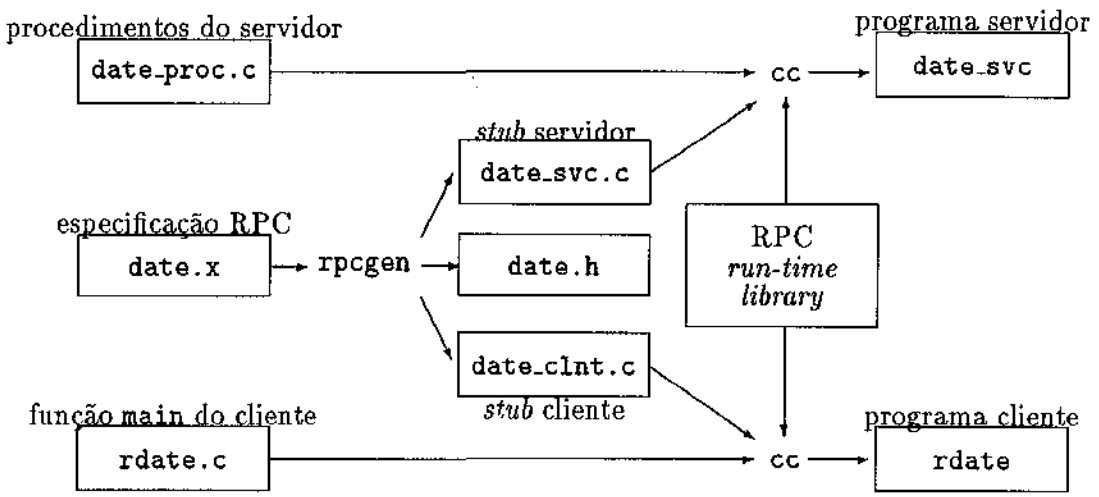



Figura 6.1: Geração de uma aplicação fazendo uso de Sun RPC

**Protocolo RPC e sua implementação no escopo da camada de redirecionamento**

Como vimos, o protocolo RPC não é uma implementação. Por ser genérico, o mesmo pode ser implementado no núcleo de um sistema operacional e/ou como um conjunto de rotinas em uma biblioteca fora do contexto do sistema operacional.

Mas, para ter um serviço de rede integrado e único, é conveniente que se faça a implementação de *portmapper* no sistema operacional, porém, não necessariamente no núcleo do sistema. O protocolo RPC e os filtros XDR podem ser implementados através de rotinas disponíveis ao programador, através de bibliotecas. A linguagem *rpcgen* foi concebida para integrar todo o ambiente composto por XDR, protocolo RPC e *portmapper*, de tal forma que detalhes de implementação sejam escondidos do programador.

A função $f_L$ está representada por *rpcgen*. *Rpcgen* se comporta como um pré-processador, inserindo e modificando o código fonte para ser submetido ao compilador. Os demais serviços podem ser implementados tanto pela função $f_L$ quanto pela função $f_{SO}$.

Este conjunto de protocolos enfocados nesta seção mostra-nos claramente a dimensão da questão de distribuição de código. Propõem-se que estes serviços sejam oferecidos como parte integrante de um conjunto de protocolos de redes. Se estes protocolos forem implementados diretamente pelo sistema operacional, ganhos substanciais podem advir disto no que diz respeito à eficiência e à homogeneização do serviço na rede. Caso contrário, pode-se

ter tais serviços limitados a uma determinada aplicação, como por exemplo, um banco de dados.

O mecanismo de mensagens e o tipo atribuído a eles, através dos filtros XDR e o mecanismo de *portmapper*, perfazem a função de um "ligador". Esta ligação "remota" em tempo de execução possibilita explorar distribuição introduzindo novos serviços, a exemplo da troca de configuração em tempo de execução do sistema Conic.

## 6.4 Conclusão

Neste capítulo exploramos o sistema Conic e o protocolo Sun RPC e sua implementação.

O Sistema Conic mostra-se interessante, entre outros aspectos, porque a função $f_L$ possui um programa "ligador", que tem a função de conectar os *ports* dos módulos, cuja conexão pode ser alterada em tempo de execução por $f_{SO}$. Isto nos leva ao conceito de "ligação dinâmica" e mostra uma desejável interação entre $f_L$ e $f_{SO}$. A função $f_{SO}$ consegue lidar com os *ports* dos módulos (e de suas instâncias).

O protocolo Sun RPC e sua implementação mostram-se importantes, visto que mecanismos de prover distribuição de código foram implementados como serviços de uma rede de comunicação, sendo possível interligar vários computadores, com sistemas operacionais diferentes, arquiteturas diferentes e geograficamente distantes entre si, cooperando para a realização de alguma tarefa. Várias linhas de pesquisa nesta direção se dão principalmente sob o modelo de referência ISO-OSI. Tais protocolos não determinam a implementação dos protocolos em $f_L$ e/ou $f_{SO}$, abstraindo destas questões. Na implementação analisada, se faz uso de rotinas *stubs*, geradas por *rpcgen*. O mecanismo de *portmapper* possibilita que em cada nodo da rede se associe um *port* a um processo passível de execução por um cliente remoto. O *port* de um processo remoto, juntamente com a identificação do nodo é o endereço unívoco do processo na rede. Em essência, existe também uma ligação dinâmica entre o processo remoto e o processo cliente. O endereço do processo remoto é resolvido em tempo de execução, fazendo uso, em nosso exemplo, do mecanismo de *portmapper*.

Apesar de Conic e o protocolo Sun RPC serem totalmente distintos, ambos enfatizam uma conexão entre $f_L$ e $f_{SO}$ e ambos implementam mecanismos de "ligação dinâmica" entre processos, pois o "endereço" do processo só é definido em tempo de execução.

# Capítulo 7

# Conclusões

Este capítulo tem o objetivo de analisar toda a camada de redirecionamento a partir dos vários exemplos e conclusões explanados nos capítulos anteriores.

As conclusões levantadas nesta dissertação possibilitaram especificar funcionalmente uma camada de redirecionamento enfatizando importantes agentes de redirecionamento discutidos em todos os capítulos.

## 7.1 Camada de Redirecionamento

A camada de redirecionamento é uma abstração, um paradigma, que introduz uma taxionomia que define quem são os agentes que provêem distribuição em um sistema e como estes agentes atuam. O enfoque desta dissertação foi dado na distribuição de código, porém a camada de redirecionamento pode se adequar perfeitamente à análise de outras características do sistema, como distribuição de dados.

A camada de redirecionamento é importante pelo próprio contexto que a distribuição de código e dados se impõe na tecnologia de *hardware* e *software* existente hoje. Dois computadores interconectados por uma rede de alta velocidade, atualmente, podem ter sua taxa de transferência maior ou igual à taxa de transferência de seus dispositivos de entrada e saída como discos magnéticos ou ópticos. Conforme a tecnologia usada, possivelmente pode-se interconectar dois processadores através de uma rede de alta velocidade e ter o mesmo desempenho de dois processadores compartilhando o mesmo barramento de dados. Concomitantemente existe uma crescente tecnologia onde detalhes de *hardware* e até mesmo de *software* são cada vez mais invisíveis para o usuário final, tornando possível a cooperação de subsistemas compostos por processadores, arquiteturas e sistemas operacionais heterogêneos. A camada de redirecionamento é uma poderosa ferramenta para a análise desta tecnologia pertinente em qualquer ambiente onde o conceito de concorrência entre processos é implementado, seja através de um sistema operacional multitarefa, seja através da interconexão de vários computadores heterogêneos ou de outros mecanismos.

Dentro da camada de redirecionamento, foi enfatizada a distribuição de código no escopo de um sistema multicomputacional, isto é, um sistema onde cada processador (ou grupo de processadores) é associado a um núcleo do sistema operacional. Basicamente tais processadores são interconectados através de uma rede de comunicação.

Além da explanação da camada de redirecionamento no capítulo 3, nos capítulos 4, 5 e 6 foram explicitados mecanismos baseados no redirecionamento forte pela função $f_L$, pela função $f_{SO}$ e pela interação de ambas, respectivamente. Das conclusões presentes em cada capítulo, podemos acrescentar:

- A partir de $f_L$:
  - A implementação da camada de redirecionamento exclusivamente pela função $f_L$, a priori, não satisfaz como um mecanismo genérico, pois basicamente apenas os objetos gerados pela mesma função se reconhecem.
  - Mecanismos de redirecionamento onde explicita-se direta e estaticamente a interconexão dos processadores penalizam a troca de configuração do sistema. O conceito de configuração do sistema foi introduzido no capítulo 4.
  - A codificação de um "objeto computável" (veja capítulo 3) deve ser independente do ambiente onde ele será executado. A codificação de um objeto deve ser independente da configuração do sistema e/ou de como este objeto será implementado.
  - A geração de objetos executáveis otimizados para um determinado ambiente é desejável. Por exemplo, um objeto que é apenas invocado localmente pode e deve ter sua implementação diferente de um objeto passível de execução remota, caso o sistema operacional não trate de uma maneira uniforme objetos locais e remotos. Porém, é também desejável que a declaração de qualquer objeto, no sistema, tenha a mesma sintaxe, independentemente de como o objeto será implementado.

- A partir de $f_{SO}$:
  - A implementação da camada de redirecionamento pela função $f_{SO}$ proporciona um mecanismo uniforme onde os objetos do sistema compartilham serviços comuns, de tal forma que os objetos gerados por qualquer função $f_L$ se reconhecem.
  - A implementação de primitivas pertinentes à camada de redirecionamento no núcleo do sistema operacional possibilita a criação de um ambiente onde "distribuição" é tratada de uma maneira uniforme e eficiente pelo sistema. Extensibilidade nestes sistemas é facilmente encontrada, agregando serviços ao sistema a partir das primitivas disponíveis no núcleo, mas sem implementar estes serviço dentro do núcleo.

- A partir de $f_{SO}$ e $f_L$:
  - É possível definir protocolos que podem ser agregados ao sistema operacional, através de biblioteca e/ou pequenos servidores que permitem implementar a camada de redirecionamento como serviços de uma rede de comunicação.
  - Mesmo em sistemas onde os principais mecanismos de redirecionamento estão implementados no sistema operacional, se faz generoso uso de compiladores e pré-processadores de código, no sentido de esconder detalhes de distribuição

do programador, possibilitando o desenvolvimento de programas passíveis de execução remota da mesma forma que se desenvolve programas para execução local.

- Sistemas que foram concebidos visando a integração $f_L$ e $f_{SO}$ implementam uma robusta camada de redirecionamento. Faz-se necessário que o sistema operacional conheça detalhes dos objetos executáveis. Assim, o sistema operacional pode, em tempo de execução, interagir com os módulos de um programa.

## 7.2 Principais Agentes de Redirecionamento

Analisamos, no decorrer desta dissertação, vários agentes (mecanismos) pertinentes à camada de redirecionamento. Alguns mecanismos foram idealizados visando estender os mecanismos existentes em um sistema computacional tradicional, que não implementa a camada de redirecionamento. Outros, por sua vez, idealizaram tais mecanismos a partir do reprojeto de várias entidades pertinentes à camada de redirecionamento. Sob este aspecto podemos classificar o projeto de uma camada de redirecionamento como evolucionário ou revolucionário, a saber:

- **Revolucionário** Consideramos sistemas que projetaram a camada de redirecionamento sob um enfoque revolucionário como sendo todos os sistemas que contemplam fortemente a camada de redirecionamento de tal forma a abandonar antigas relações entre $f_L$ e $f_{SO}$ através de um novo projeto dos mecanismos de redirecionamento combinando ambas funções. A prioridade em tais sistemas é explorar a forte interação entre $f_L$ e $f_{SO}$, implicando inclusive na criação de novos paradigmas. Um exemplo de projeto revolucionário é o sistema Conic e sua linguagem de configuração.

- **Evolucionário** Consideramos sistemas que projetaram a camada de redirecionamento sob um enfoque revolucionário como sendo os sistemas que implementam a camada de redirecionamento estendendo os mecanismos existentes nos sistemas computacionais que possuem uma fraca, ou praticamente não tinham nenhuma camada de redirecionamento. Um exemplo de projeto evolucionário é o sistema Mach como uma extensão do Unix.

Essa dicotomia é pertinente em nossa discussão visto ser mais um parâmetro que estabelece o comprometimento da camada de redirecionamento com todo o sistema.

Iremos discutir abaixo os mais relevantes mecanismos de redirecionamento identificados nesta dissertação, partindo do conceito de objetos e acrescentando novos mecanismos até que no final tenhamos idealizado uma poderosa camada de redirecionamento. Assim, iremos delinear uma camada de redirecionamento baseada nas principais conclusões tiradas em toda a dissertação.

### Unidade de distribuição: Objetos

Objetos, por encapsularem os dados e os procedimentos que definem seu comportamento em uma entidade única, mostram-se adequados para serem adotados como unidade de

distribuição em um sistema multicomputacional. Tal escolha é justificada principalmente pelas seguintes razões:

- Conceitualmente, objetos apenas sincronizam e comunicam entre si através de troca de mensagens.
- A declaração de um objeto pode resultar em várias implementações, onde cada implementação preserva a semântica do objeto.
- Como os dados e o código (procedimentos) do objeto são tratados como uma entidade única, os objetos possuem uma modularidade que, corretamente explorada, proporcionam uma grande flexibilidade ao sistema. Por exemplo, no paradigma de objetos a instanciação de uma variável (no sentido de linguagens procedimentais) implica também na instanciação do código (procedimentos) que irá atuar sob estas variáveis, possibilitando uma flexibilidade potencial quanto ao local onde o objeto poderá ser instanciado, sugerindo uma independência de localidade.
- Pode-se facilmente prover mecanismos de proteção contra acessos de outros objetos não autorizados, sem nenhuma alteração do modelo, a exemplo do mecanismo de *capabilities* discutido nos capítulos anteriores.
- O modelo de objetos é adequado para o nível de granularidade desejado em um sistema multicomputacional, variando de uma média granularidade a uma grossa.
- Objetos possuem uma interface bem definida que estabelece quais as mensagens que poderão ser enviadas ao objeto. Sob outro enfoque, esta interface define um conjunto de protocolos que o objeto irá respeitar para que se estabeleça um relacionamento entre ele e outro objeto. Esta interface possibilita também associar *ports* de saída a cada elemento da interface que especifica o envio de mensagens e associar *ports* de entrada a cada elemento da interface que estabelece o recebimento de mensagens.

Assim, a nossa unidade de concorrência e distribuição será um objeto.

### Instanciação de um objeto

Partindo do pressuposto que em nosso sistema tudo é objeto, incrementaremos o nosso sistema computacional conferindo a ele um mecanismo que o possibilite instanciar objetos em qualquer nodo. Para que isto seja possível, devemos ter claro que a instanciação de um objeto implica, no mínimo:

- Na alocação de área de memória para os dados que estão associados aos objeto em sua especificação,
- Na alocação de recursos com o objetivo de permitir que os métodos (procedimentos) associados ao objeto possam ser executados.

Assim, faz-se necessário alocar área de memória suficiente para que seja possível a instanciação de um objeto. Um objeto pode, durante sua instanciação, requisitar a alocação

da área de dados e código ao sistema operacional. Tal objeto é potencialmente independente da localidade (nodo) onde será instanciado. Por outro lado, uma grande área pode ter sido alocada e a instanciação de um objeto pode ter sido feita através de mecanismos embutidos no próprio objeto, sem fazer uso direto do sistema operacional. Tal metodologia é apropriada quando se tem pequenos objetos que são agregados a um objeto maior. Esse objeto maior é responsável em alocar os recursos do sistema operacional e gerenciar a utilização destes recursos com os demais objetos agregados a ele. Denominaremos esse objeto maior como "objeto global", que pode ter nenhum ou vários objetos locais associados a ele.

Baseado na explanação acima, podemos concluir:

- É possível ter objetos locais a outros objetos, onde a instanciação do objeto global implica na instanciação dos objetos locais agregados a ele.
- Os objetos globais são tratados como uma unidade básica de distribuição, visto que através dele todos os recursos são alocados do sistema. Assim, a princípio, um objeto global pode ser instanciado em qualquer nodo do sistema.

Tal discussão converge para o problema da granularidade dos objetos manipulados pelo sistema operacional. Pelas responsabilidades impostas a um sistema operacional torna-se oneroso tratar todos os objetos como globais. Também, em um sistema multicomputacional composto por vários nodos, onde cada nodo possui um grande número de objetos globais a todo o sistema, é intrínseco uma sobrecarga ao sistema pela quantidade de referências que devem ser gerenciadas.

Retornado ao projeto da camada de redirecionamento, torna-se clara a necessidade da implementação de, no mínimo, dois tipos de objetos: locais e globais. Também é importante que todos os objetos, globais ou locais, sejam declarados de uma maneira única de tal forma que a semântica associada ao objeto seja a mesma independentemente de como o objeto será gerado ou manipulado pelo sistema.

### Interface, Ports e Ligação Dinâmica

Uma interface de um objeto define as mensagens que são enviadas e recebidas pelo objeto. Sistemas complexos são comumente desenvolvidos em módulos. Cada módulo é compilado individualmente, de tal forma que as referências a objetos localizados em outros módulos ficam pendentes para serem resolvidas pelo programa ligador. Um programa ligador resolve as referências (endereços) que não podem ser resolvidas por ocasião da compilação do módulo. Um endereço de um objeto em um sistema multicomputacional lida com a identificação unívoca deste objeto na rede. Cada elemento da interface do objeto é conceitualmente um *port*. Assim, a interface de um objeto identifica os *ports* do objeto.

Um importante agente de redirecionamento pode ser implementado explorando as referências a objetos. A comunicação entre dois objetos implica na conexão de seus *ports*, estabelecendo um canal de comunicação. Iremos descrever este importe mecanismo através de um exemplo simples. Imaginemos um objeto servidor de impressão. Uma das mensagens possíveis de ser recebida por ele é a mensagem:

`imprima` *mensagem_a_ser_impressa*

Quando o servidor imprimir a *mensagem_a_ser_impressa* este deverá retornar ao objeto que o invocou uma mensagem de reconhecimento. Por algum motivo, como por exemplo a atualização de versão, desejamos trocar o objeto servidor de impressão. Porém é desejável apenas trocar um objeto sem desativar todos os demais, isto é, sem reiniciar todo o processamento para que apenas um objeto seja trocado. Com a implementação de dois mecanismos pertinentes à camada de redirecionamento isto é possível:

- Qualquer referência a um objeto deve ser feita indiretamente através de um objeto intermediário, como o mecanismo de caixas postais descrito no capítulo 2. Desta forma, todos os objetos clientes que referenciarem o objeto servidor de impressão referenciarão a caixa postal correspondente. A troca do objeto que implementa o servidor de impressão se fará apenas trocando o objeto que será referenciado pela caixa postal como servidor de impressão. É também necessário implementar tais caixas postais de tal forma que seja possível a troca de endereço do objeto em tempo de execução.

- Garantir que não existe processamento pendente em um dado objeto. Uma das possíveis implementações para identificar se existe algum processamento pendente, pode ser baseada na monitorização da entrada e saída das mensagens nos *ports*, associando contadores a cada *port* ou grupo de *ports*, de tal forma que tais contadores assegurem que não existe nenhum processamento em andamento no objeto. Por exemplo, cada mensagem `imprima` pode incrementar um contador e cada mensagem de reconhecimento decrementa este mesmo contador. Assim, se o contador estiver com zero, significa que não existe nenhum processamento pendente no objeto.

Assim, o utilitário que fará a troca dinâmica (em tempo de execução) deverá, no minímo:

- Instanciar o novo objeto servidor de impressão,
- Trocar o endereço do objeto servidor de impressão na caixa postal,
- "Matar" o antigo objeto servidor de impressão quando não existir nenhum processamento pendente no objeto, isto é, quando o objeto mudar para o estado passivo.

Apesar de desconsiderada aqui a possibilidade de *deadlock*, o relacionamento múltiplos clientes-múltiplos servidores e a checagem da interface dos objetos (assegurando que o novo objeto será compatível com o objeto antigo), o mecanismo explanado possibilita a troca de objetos em tempo de execução, além de possibilitar ligação dinâmica entre os objetos. Uma variação deste mecanismo foi implementada no sistema Conic, explanado no capítulo 6.

### Identificação e endereçamento de um objeto

Um endereço de um objeto em um sistema multicomputacional identifica univocamente este objeto na rede. Assegurando que cada instância de um objeto possui um identificador único no nodo, estender esta identificação para todo o sistema significa adicionar o identificador do nodo ao identificador do objeto.

Um importante mecanismo pertinente à camada de redirecionamento é um servidor de nomes de objetos. Tal mecanismo pode estar incluso no mecanismo de caixas postais

sugerido no item acima. O objetivo de se ter um servidor de nomes é proporcionar ao sistema uma associação entre o nome do objeto invocado pelo processo cliente e o objeto que realmente será invocado, possibilitando que objetos sejam referenciados independentemente de sua localidade. Imaginemos um objeto servidor de impressão. Se vários objetos clientes invocarem o objeto servidor especificando sua localidade física, caso o objeto servidor de impressão seja instanciado em outra localidade, a priori, todo os objetos que referenciam este objeto servidor deverão ser modificados, recompilados, religados e todos os objetos pertencentes ao programa deverão ser reinstanciados para refletir esta nova situação.

Uma possível implementação de servidor de nomes poderia basear-se na seguinte estratégia, semelhante a usada no mecanismo *portmapper* (explanado no capítulo 6):

- Todos os objetos que desejam ser considerados globais para todo o sistema devem ser registrados no servidor de nomes. Cada nodo do sistema possui apenas o seu servidor de nomes.
- Todo o objeto que deseja invocar algum objeto e desconhece sua localidade, pergunta aos servidores de nomes de cada nodo se tal objeto está registrado. Se estiver registrado o servidor retorna com a identificação do objeto desejado, que agora pode ser invocado diretamente.

### Sistema de execução

Em uma camada de redirecionamento implementada sob um enfoque revolucionário, há uma integração entre especificação de um objeto e a implementação e suporte deste objeto no sistema, sendo estes últimos atribuídos geralmente ao sistema operacinal. Por outro lado, uma camada de redirecionamento implementada sob um enfoque evolucionário implica na necessidade de estender todo o sistema para suportar as abstrações implementadas pela camada. Assim, pela complexidade das funções implementadas, muita vezes se faz necessário que em cada nodo esteja presente um pequeno sistema de execução[1] que suporte as funções implementadas pela camada de redirecionamento.

Em nosso esboço da camada de redirecionamento, identificamos o mecanismo de caixa postal, ligação dinâmica e servidor de nomes como possíveis candidatos a uma implementação baseada em um sistema de execução. Apesar do sistema de execução pertencer à camada de redirecionamento, como dissemos, este poderá não existir, caso implementado sob a função $f_{SO}$. As vantagens e desvantagens de implementar um determinado mecanismo pertinente à camada de redirecionamento nas funções $f_L$ e $f_{SO}$ foram exaustivamente analisadas nesta dissertação.

A construção de uma camada de redirecionamento baseada nos mecanismos descritos acima certamente proporcionará um ganho de flexibilidade e funcionalidade ao sistema. Implementar tais mecanismos sem o comprometimento do sistema operacional poderá ocasionar uma elevada sobrecarga ao sistema.

[1] Em inglês: *Run Time*

## 7.3 Considerações Finais

Esta dissertação possui o objetivo de enfocar como é feita distribuição de código em um sistema multicomputacional. A camada de redirecionamento e o modelo computacional no qual foi definido essa camada foram ferramentas que viabilizaram o inter-relacionamento de assuntos aparentemente distintos como: a linguagem DP, o sistema operacional Amoeba, a linguagem de configuração Conic, o protocolo Sun RPC e redes de comunicação.

Houve uma preocupação constante para que a explanação de linguagens, sistemas operacionais e protocolos, contidos nas seções "Análise de Casos" dos capítulos 4, 5 e 6, possibilitassem uma visão geral dos "casos" em análise, sem uma grande sobrecarga de detalhes. Os "casos" foram cuidadosamente escolhidos para serem realmente distintos uns dos outros, de tal forma que um produto pudesse retratar uma gama de similares. Por exemplo, escolheu-se Sun RPC entre Apollo RPC e Courier RPC [Stevens90]. Assim o objetivo não foi explanar sobre todos os possíveis produtos que exploram distribuição, mas foi explanar sobre os mais significativos, dentro do contexto da dissertação.

Foi inevitável a abordagem de uma grande quantidade de tópicos no capítulo 2, visto que os mesmos definem os fundamentos de toda dissertação e a nomeclatura usada. Por causa disto, em cada introdução de um tópico naquele capítulo é explanado como o mesmo é subdividido.

A camada de redirecionamento é um paradigma que pode ser aplicado em diversos outros aspectos da computação distribuída, como distribuição de dados, ou até mesmo para análise de outras ferramentas que não possuem relação com distribução, a exemplo da análise das diversas implementações de processos leves (*threads*) em sistemas derivados do UNIX. Porém, a nossa discussão foi restrita à distribuição de código, inclusive para maior clareza na explanação do modelo computacional e da camada de redirecionamento. Futuros trabalhos podem explorar estes assuntos, sob a ótica da camada de redirecionamento e o seu modelo computacional.

Abaixo, iremos relacionar algumas contribuições deste trabalho:

- Integra de uma forma coerente diversos tópicos como linguagens, sistemas operacionais, redes de comunicação e outras ferramentas, em um paradigma único.
- Estabelece as limitações ao se implementar um determinado serviço através de linguagens, através do sistema operacional, ou de alguma outra solução híbrida.
- Estabelece uma taxionomia quanto à distribuição de código em um sistema multicomputacional, ao se aplicar a camada de redirecionamento dentro desse contexto.
- Provê um novo novo paradigma para a análise de sistemas distribuídos baseados na camada de redirecionamento e no seu modelo computacional.
- Possibilita aplicar o paradigma camada de redirecionamento em diversos contextos, inclusive em contextos que desconsiderem a distribuição de código.

Quanto às limitações deste trabalho, podemos citar o tratamento simplificado das questões de proteção e segurança do sistema, mesmo tendo sido relativamente bem explanadas o mecanismo de *capabilities*.

# Bibliografia


[AccBar86] Accetta, M., Baron R., Golub, D. Rashid, R., Tevanian A. e Young M. *Mach: A new Kernel Foundation for UNIX Development* Proc. Summer 1986 USENIX Technical Conference and Exhibition, Junho 1986.

[AndSch83] Andrews, G.R. e F.B. Schneider *Concepts and Notations for Current Programming* ACM Computing Surveys, vol. 15, num. 1 (Março 1983), pag. 3-47.

[Andrews91] Andrews, G. R. *Paradigms for Process Interaction in Distributed Programs* ACM Computing Surveys, vol. 23, num. 1 (Março 1991), pag. 49-90.

[ASN.1] *Abstract Syntax Notation 1 - ASN.1* IEEE International Standart 8824.

[BallST89] Bal, H. E., Steiner, J.G. e Tanenbaum, A. S. *Programming Languages for Distributed Computing Systems* ACM Computing Surveys, vol. 21, num. 3 (Setembro 1989), pag. 261-321.

[BenAri90] Ben-Ari *Principles of Concurrent and Distributed Programming* Prentice Hall, New Jersey, 1990.

[Black86] Black, A., Hutchinson N., Jul, E. e Levy H. *Object Structure in the Emerald System* OOPSLA'86 Proceedings - ACM Sigplan Notices vol. 21, num. 11 (Setembro 1986), pag. 78-86.

[BirNel84] Birrell, A. e Nelson, B. *Implementing Remote Procedure Call* ACM Transactions on Computer Systems, vol. 2, num. 1 (Fevereiro 1984), pag. 39-59.

[Brinch78] Brinch Hansen *Distributed Process: A Concurrent Programming Concept* CACM, vol. 21, num. 11 (Novembro 1978), pag. 934-941.

[Cheriton84] Cheriton, D. *The V Kernel: A Software Base for Distributed Systems* IEEE Software, vol. 1, num. 4 (Abril 1984), pag. 19-42.

[ChinChan91] Chin, R. S. e Chanson, S. T. *Distributed Object-Based Programming Systems* ACM Computing Surveys, vol. 23, num. 1 (Março 1991), pag. 91-124.

[Cook80] Cook, R. **MOD - A Language for Distributed Programming* IEEE Transactions on Software Engineering, vol. SL-6, num. 6 (Novembro 1980), pag. 563-571.

[Dennis66] Dennis, J. B. e Van Horn, E. C. *Programming semantics for multiprogrammed computations* CACM, vol. 9, num. 3 (Março 1966), pag. 143-155.

[Dijkst75] Disjkstra, E.W. *Guarded Commands, Nondeterminininacy and Formal Derivation of Programs* CACM, vol. 18, num. 8 (Agosto 1975), pag. 453-457.

[Frenkel93] Frenkel, K. *An Interview with Robin Milner* CACM, vol. 36, num. 1 (Janeiro 1993), pag. 90-97.

[Gehani84] Gehani, N. H. *Broadcasting Sequential Process (BSP)* IEEE Trans. on Software Enginerring, vol. SE-10, num. 4 (Julho 1984), pag. 343-351.

[GolRob83] Goldberg, A. e Robson, D. *Smalltalk-80: The Language and its Implementation* Addison-Wesley, Mass, 1983.

[Hoare74] Hoare, C.A.R. *Monitors: An Operating System Structuring Concept* CACM, vol. 17, num. 10 (Outubro 74), pag. 549-557.

[Hoare78] Hoare, C. A. R. *Communicating Sequential Process* CACM, vol. 21, num. 8 (Agosto 1978), pag. 666-677.

[HwaBri85] Hwang, K. e Briggs, F. *Computer Architecture and Parallel Processing* McGraw-Hill,1985.

[iPSC87] Intel Corporation *iPSC Concurrent Debugger Manual, 2nd. Edition* Março de 1987.

[JoRas86] Jones, M. B. e Rashid, R. F. *Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems* OOPSLA'86 Proceedings-ACM Sigplan Notices vol. 21, num. 11 (Setembro 1986), pag. 67-77.

[JulHut88] Jul, E., Hutchinson, N. e Black, A. *Fine-Grained Mobility in the Emerald System* ACM Transactions on Computer Systems, vol. 6, num. 1 (Fevereiro 1988), pag. 109-133.

[KraMag83] Kramer, J., Magge, J., Sloman, M. e Lister, A. *CONIC: an Integrated Approach to Distributed Computer control systems* IEE Proc. vol. 130, Pt. E, num. 1 (Janeiro 1983), pag. 1-10.

[KraMag85] Kramer, J. e Magge J., *Dynamic Configuration for Distributed Systems* IEEE Transactions on Software Engineering, vol. SE-11, num. 4 (Abril 1985), pag. 424-436.

[Liskov84] Liskov, B. *Overview of the Argus Language and System* Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, MIT, Cambridge, Mass, Fevereiro de 1984.

[MulTan86] Mullender, S. e Tanenbaum, A. *The Design of a Capability-Based Distributed Operating System* The Computer Journal, vol. 29, num. 4, (Abril 1986), pag. 289-299.

[Nelson81] Nelson, B. *Remote Procedure Call* Report CSL-81-9, Xerox Palo Alto Research Center, Maio 1981.

[Parnas72] Parnas, D.L. *On the Criteria To Be Used in Decomposing Systems into Modules* CACM, vol. 15, num. 12 (Dezembro 1972), pag. 1053-1058.

[Postel80] Postel,J. *User Datagram Protocol* DARPA Internet Program Protocol Specification RFC 768, DARPA Information Science Institute, Agosto 1980.

[Postel81] Postel,J. *Transmission Control Protocol* DARPA Internet Program Protocol Specification RFC 793, DARPA Information Science Institute, Setembro 1981.

[Raynal86] Raynal, Michel *Algorithms for Mutual Exclusion* The MIT Press, Cambridge, Mass, 1986.

[Raynal88] Raynal, Michel *Distributed Algorithms and Protocols* John Wiley & Sons, 1988.

[Ritchie74] Ritchie, D.M. e Thompson, K. *The UNIX Time-Sharing System* CACM, vol. 17, num. 7 (Julho 1974), pag. 365-375.

[SocoKale91] Socolofsky, T. e Kale, C. *A TCP/IP Tutorial* RFC1180, DARPA Information Science Institute, Janeiro de 1991.

[Stevens90] Stevens, W. *UNIX Network Programming* Prentice Hall, New Jersey, 1990.

[SUN88] Sun Microsystems, Inc. *RPC: Remote Procedure Call - Version 2* RFC 1057, DARPA Information Science Institute, Junho 1988.

[SUN87] Sun Microsystems, Inc. *XDR: External Data Representation Standard* RFC 1014, DARPA Information Science Institute, Junho 1987.

[Tanen89] Tanenbaum, A. *Computer Networks - 2a. edition* Prentice Hall, New Jersey, 1989.

[TanRe90] Tanenbaum, A. e van Renesse, R. *Distributed Operating Systems* ACM Computing Surveys, vol. 17, num. 4 (Dezembro 1985), pag. 419-470.

[TanRe90] Tanenbaum, A., van Renesse, R., van Staveren, H., Sharp, G.J., Mullender, S., Jansen, J. e van Rossum, G. *Experiences with the Amoeba Distributed Operating System* CACM, vol. 33, num. 12 (Dezembro 1990), pag. 46-63.

[TanDon88] Tangney, B. e O'Mahony, D. *Local Area Networks and their Applications* Prentice Hall, New Jersey, 1988.

[TayAn90] Tay, B.H. e Ananda, A.L. *A Survey of Remote Procedure Calls* ACM Operating Systems Review vol. 24, num. 3 (Julho 1990), pag. 68-79.

[WegSmo83] Wegner, P and Smolka, S. A. *Process, Tasks, and Monitors: A Comparative Study of Current Programming Primitives* IEEE Transactions on Software Engineering, vol. SE-9, num. 4 (Julho 1983), pag. 303-360.

[Wegner90] Wegner, P. *Concepts and Paradigms of Object-Oriented Programming* ACM OOPS Messenger, vol. 1, num. 1 (Agosto 1990), pag. 7-87.

[Whidde87] Whiddett, D. *Concurrent Programming for Software Engineers* Ellis Horwood Limited, West Sussex, England, 1987.

[Wirth77] Wirth, N. *Modula: A Language for Modular Multiprogramming* Software-Practice and Experience vol. 7, (Julho 1977), pag. 3-35.

[Zimmer80] Zimmermann, H. *OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection* IEEE Transactions on Communications, vol. COM-28, num. 4 (Abril 1980), pag. 425-432.