UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Thiago Robert Claudino dos Santos

Um Sistema de Comunicação Configurável e Extensível Baseado em Metaprogramação Estática

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Antônio Augusto Medeiros Fröhlich

Florianópolis, Fevereiro de 2006

Um Sistema de Comunicação Configurável e Extensível Baseado em Metaprogramação Estática

Thiago Robert Claudino dos Santos

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Computação Paralela e Distribuída e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Dr. Raul Sidnei Wazlawick

Banca Examinadora

Orientador: Prof. Dr. Antônio Augusto Medeiros Fröhlich

Prof. Dr. Mário Antonio Ribeiro Dantas

Prof. Dr. Benhur de Oliveira Stein

Prof. Dr. Carlos Barros Montez

"I always knew C++ templates were the work of the Devil, and now I'm sure. :-)"
– Cliff Click

"Whatever language you write in, your task as a programmer is to do the best you can with the tools at hand. A good programmer can overcome a poor language or a clumsy operating system, but even a great programming environment will not rescue a bad programmer."
– Kernighan and Pike

"After the game the king and the pawn go in the same box."
– Italian proverb

Às duas mulheres mais importantes da minha vida: minha esposa Katherine Anne Casey e minha mãe Inez Maria de Fátima Robert.

Agradecimentos

Considero essa dissertação o ponto culminante da minha formação acadêmica em ciências da computação pela *Universidade Federal de Santa Catarina* (UFSC), onde cursei a graduação e o programa de mestrado ao longo de sete anos. Muitas pessoas contribuiram direta ou indiretamente com essa formação e, mais especificamente, com o desenvolvimento dessa dissertação e, correndo o risco de deixar de mencionar algumas dessas pessoas, eu gostaria de fazer alguns agradecimentos.

Primeiramente eu gostaria de agradecer aos membros da minha família, principalmente a minha mãe *Inez Maria de Fátima Robert* e a minha esposa *Katherine Anne Casey*. Conheci a Katie dois meses após a minha formatura da graduação, logo antes de começar o mestrado, e ela pode acompanhar os altos e baixos dos quase três anos ao longo dos quais eu desenvolvi o trabalho descrito nessa dissertação. Sem o seu apoio e compreensão eu tenho certeza de que não teria sido tão bem sucedido nessa empreitada. Minha mãe sempre apoiou as minhas decisões, mesmo algumas das mais controversas, e sempre me incentivou a desafiar os meus próprios limites e buscar excelência em tudo o que eu fiz. Através de seus exemplos de caráter e lições de moral ela guiou a minha formação como profissional e, mais importante, como pessoa e sem a sua ajuda eu não teria condições de realizar as várias conquistas das quais eu tanto me orgulho hoje.

Muitas pessoas consideram os amigos como uma segunda família, a família que você tem a oportunidade de escolher. Eu gostaria de agradecer de maneira geral a todos os meus amigos pessoais, os quais eu muitas vezes tive que deixar de lado durante o desenvolvimento desse trabalho. As amizades são fundamentais para o desenvolvimento de uma personalidade equilibrada e eu posso dizer que eu tive a sorte de, ao longo dos anos, poder contar com as melhores amizades que eu poderia ter escolhido.

O trabalho de pesquisa descrito nessa dissertação faz parte de um contexto maior: os diversos projetos de pesquisa desenvolvidos pelo *Laboratório de Integração de Hardware e Software* (LISHA) da UFSC. O LISHA é um dos laboratórios mais antigos do *Departamento de Informática e Estatística* (INE) da UFSC e ao longo de seus quase quinze anos de história diversos estudan-

tes de graduação e mestrado ajudaram a construir uma reputação que hoje é reconhecida a nível nacional. Eu gostaria de agradecer a todos os membros do LISHA com os quais eu tive o prazer de trabalhar e que desempenharam um papel ativo e fundamental no desenvolvimento dessa dissertação. Em especial, agradeço ao professor *Antônio Augusto Medeiros Fröhlich*, o Guto, meu orientador no mestrado e atual responsável pelo LISHA, por todo o apoio e pelas lições ensinadas dentro e fora da sala de aula.

Por último, mas não menos importante, gostaria de agradecer ao corpo docente, funcionários e alunos da UFSC que tanto me ensinaram ao longo dos sete últimos anos. Durante o convívio diário na UFSC essas pessoas me ajudaram a desvendar os mistérios relacionados as ciências da computação e a pesquisa científica em geral e desempenharam um papel fundamental, através de exemplos e contra-exemplos, na minha formação profissional. Gostaria de agradecer mais especificamente ao INE e a Pós-Reitoria de Pós-Graduação pelo incentivo financeiro que viabilizou a minha participação na conferência HPCS 2005 e a equipe do Labsoft onde eu tive a oportunidade de trabalhar durante parte da minha graduação e mestrado.

Eu gostaria também de mencionar que o trabalho de pesquisa e desenvolvimento descrito nessa dissertação não poderia ter sido realizado sem o conjunto de ferramentas desenvolvidas pela comunidade *Open Source*. Tenho muito orgulho de poder dizer que todas as ferramentas utilizadas no desenvolvimento desse texto e na implementação do sistema de comunicação descrito nos próximos capítulos são frutos de projetos *Open Source* e espero poder utilizar o conhecimento adquirido com o desenvolvimento desse trabalho para contribuir com esse movimento em breve.

Resumo

Sistemas computacionais de alto desempenho exercem hoje um papel fundamental na pesquisa científica e na indústria de tecnologia. Simulações em larga escala, tais como as utilizadas para estudar as causas e os efeitos do aquecimento global, e aplicações de processamento intensivo, como por exemplo o seqüenciamento de DNA, dependem do processamento e armazenamento distribuído de informações fornecidos por supercomputadores.

Atualmente *agregados de PCs*, supercomputadores construídos através da interconexão de computadores pessoais convencionais, apresentam desempenho comparável ao de arquiteturas como *processadores massivamente paralelos* e *computadores vetoriais*, conquistando um lugar de destaque na área de computação de alto desempenho e sendo considerados a melhor opção em infra-estrutura computacional quando leva-se em consideração a relação custo/desempenho.

O trabalho de pesquisa descrito nessa dissertação consiste no desenvolvimento de um sistema de comunicação extensível e configurável, projetado para o domínio de agregados de PCs dedicados. Esse sistema de comunicação é constituído por um conjunto de protocolos leves e um framework meta-programado responsável por prover mecanismos que permitem selecionar, configurar e combinar esses protocolos de acordo com os requisitos da aplicação. Esse paradigma oferece diversas vantagens, incluindo a habilidade de criar novos protocolos de comunicação sob demanda e permitir que aplicações experimentem diferentes configurações de protocolo, coletando métricas para identificar a melhor configuração para as suas necessidades.

Palavras-chave: metaprogramação, sistemas operacionais orientados à aplicação, protocolos de comunicação leves.

Keywords: metaprogramming, application-oriented operating systems, lightweight communication protocols.

Abstract

Today, high-performance computing can be considered ubiquitous in research centers and the high-tech industry. Large scale simulations, like the ones used to study the causes and effects of global warming, and processing-intensive applications, such as DNA sequencing, rely in the distributed processing and storage provided by supercomputers. Besides, a wide range of applications that are present in our daily activities, such as Internet portals and search engines, use high-performance computers to enable high-availability environments and to implement load-balancing solutions.

The recent availability of high-performance commodity processors and communication networks has triggered the development of *clusters of PCs*, supercomputers composed by interconnected personal computers that can achieve processing performance comparable with specialized architectures, such as *massively parallel processors* and *vector computers*. Clusters of PCs are now more widely used than any other type of parallel computer because of their low cost, flexibility and accessibility.

This thesis describes in detail the architecture of a configurable and extensible communication system designed for dedicated clusters. This communication system is composed of a set of lightweight communication protocols and a meta-programmed framework used to select, configure and combine these protocols according to the requirements dictated by the application. This paradigm offers a number of potential advantages, including the ability to combine protocols without the use of layering and the possibility of allowing that applications experiment with different communication protocols, collecting metrics in order to identify the best one for their needs. Moreover, to structure communication software in such a modular fashion enhances maintainability and extensibility, and permits that new communication protocols and services be developed on demand.

Keywords: metaprogramming, application-oriented operating systems, lightweight communication protocols.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O termo *computador* já foi utilizado pela indústria e centros de pesquisa para designar os profissionais que efetuavam os cálculos atualmente resolvidos, a taxa de milhões por segundo, pelos computadores modernos. Na década de 20 a expressão *máquina computadora* passou a ser empregada para referir-se a qualquer máquina que desempenhasse o papel de um computador humano. Entre as décadas de 40 e 50, com o advento dos computadores eletrônicos, a expressão máquina computadora foi substituída pelo termo computador, que no início era acompanhado por adjetivos como digital ou eletrônico. O físico Michio Kaku, em seu livro Visões do Futuro [1], afirma que o termo computador vai cair em desuso a medida que os PCs de hoje evoluírem, tornando-se máquinas com propósitos mais específicos. Kaku prevê que em alguns anos os computadores estarão tão integrados ao nosso ambiente quanto o papel, outra maravilha tecnológica da humanidade. Chamar o conjunto de computadores que manipularemos no nosso dia-a-dia pelo termo mais genérico computador vai ser tão inapropriado quanto chamar uma nota fiscal, um livro e uma bloco de anotações pelo termo mais genérico *papel*.

A definição do termo *supercomputador* também já mudou diversas vezes. No final da década de 80, o governo americano definiu um supercomputador como sendo um sistema computacional capaz de realizar mais do que 100 milhões de operações de ponto flutuante por segundo. Atualmente essa definição é claramente obsoleta tendo em vista que PCs modernos são capazes de realizar aproximadamente 5 bilhões de operações de ponto flutuante por segundo. De maneira mais geral, podemos definir supercomputadores como sistemas computacionais pelo menos uma ordem de magnitude mais rápidos do que o mais rápido dos PCs vendido comercialmente. O termo *supercomputação*, cunhado pelo jornal *New York World* na década de 20, ainda hoje é utilizado para denotar as várias atividades relacionadas ao projeto, desenvolvimento e utilização de

supercomputadores.

Tradicionalmente, o termo *computação de alto desempenho* (*High-Performance Computing* ou simplesmente HPC), tem sido associado ao termo supercomputação. Entretanto, com o rápido desenvolvimento das tecnologias relacionadas à computação de alto desempenho observado na última década, a associação estabelecida entre infra-estruturas computacionais de alto desempenho e supercomputadores tornou-se obsoleta. Atualmente, o estado da arte em computação de alto desempenho consiste em infra-estruturas compostas por computadores de alta capacidade, geograficamente dispersos, interconectados por redes de comunicação extremamente rápidas e escaláveis. Esses computadores compartilham entre si ciclos de processamento, espaço de armazenamento, equipamentos dedicados a colher os dados a serem analisados, tais como aceleradores de partículas ou telescópios espaciais, e até mesmo os próprios conjuntos de dados, que comumente chegam a ordem dos petabytes.

Sistemas computacionais de alto desempenho exercem um papel fundamental na maneira como a pesquisa científica é conduzida em diversas áreas, tais como bioengenharia, nanotecnologia e meteorologia. De fato, uma das mais recentes contribuições da ciência da computação para pesquisadores de outros campos é a *ciência computacional*, que, apoiada em ambientes de computação de alto desempenho, busca compreender o funcionamento de sistemas naturais através do uso e a análise de modelos matemáticos. O cientista computacional, que se junta aos tradicionais teoristas e experimentalistas, utiliza a infra-estrutura computacional disponível como uma ferramenta para construir e processar modelos matemáticos que simulam o comportamento de sistemas complexos, variando de galáxias a moléculas.

1.1 Software de Propósito Geral em Computação sobre Agregados

No mesmo ritmo em que novas tecnologias em infra-estrutura computacional promovem mudanças na maneira como a pesquisa científica é conduzida, elas também promovem mudanças na maneira como são projetados os ambientes de software que fazem a intermediação entre as aplicações e o hardware. Nos últimos anos, cientistas da computação têm presenciado uma verdadeira revolução na área de software básico, desencadeada pela constatação de que os requisitos das aplicações e ambientes de hardware modernos invalidam algumas das premissas que guiaram o desenvolvimento de sistemas operacionais e sistemas de comunicação amplamente difundidos.

A computação sobre agregados de PCs, a classe de sistemas computacionais de alto desempenho que mais cresceu nos últimos anos [2], é um dos domínios onde esse problema teve um impacto considerável.

Agregados de PCs têm se destacado bastante na área de computação de alto desempenho, sendo considerados a melhor opção em infra-estrutura computacional quando leva-se em consideração a relação custo/desempenho. Agregados são sistemas de computação paralelos compostos por uma coleção de nodos de processamento interligados por uma rede de comunicação de alto desempenho, no qual cada um dos nodos é um sistema completo, capaz de operar independentemente dos outros. Agregados podem ser montados utilizando-se nodos de processamentos e redes de comunicação disponíveis comercialmente, o que torna o sistema computacional mais acessível e permite que as inovações tecnológicas implementadas pelos fabricantes de processadores e redes de comunicação sejam rapidamente incorporadas. Os primeiros agregados de PCs, que surgiram a cerca de doze anos, utilizavam esse mesmo conceito de empregar componentes pré-fabricados também para o ambiente de software. Essa classe de agregados recebeu a denominação de Beowulf e popularizou a adoção do sistema operacional de propósito geral Linux nos nodos de processamento dos agregados. Agregados de PCs Linux ainda hoje desempenham um importante papel na área de computação de alto desempenho e no setor comercial, sendo considerados o sistema computacional paralelo mais amplamente difundido.

O principal problema com a estratégia de utilizar software de propósito geral em computação sobre agregados é que os sistemas operacionais de propósito geral, tais como o Unix, o Linux e o Windows, foram projetados sem levar em consideração os requisitos específicos das aplicações e do ambiente de hardware paralelos, o que afeta o desempenho, a usabilidade e a escalabilidade do sistema computacional. Além disso, os sistemas de comunicação disponibilizados por esses sistemas operacionais não foram projetados de maneira a tirar proveito das recentes inovações tecnológicas implementadas pelos fabricantes de redes. Implementações tradicionais de sistemas de comunicação colocam todo o processamento relacionado aos serviços de comunicação no núcleo do sistema operacional e, como conseqüência, o caminho crítico durante o envio e recebimento de mensagens inclui operações caras, tais como trocas de contexto, desencadeadas por chamadas de sistemas, e tratamento de interrupções.

De maneira a prover software de comunicação com desempenho próximo ao limite imposto pelas modernas tecnologias de rede, cientistas da computação tiveram que “tirar o sistema operacional do caminho” com os sistemas de comunicação em nível de usuário (ULC), que permitem que protocolos leves acessem diretamente as funcionalidades disponibilizadas pelo hardware de

comunicação de maneira otimizada. Como conseqüência, sistemas de comunicação modernos têm abandonado a clássica arquitetura em camadas e que guiou o desenvolvimento de protocolos bastante difundidos como os protocolos da arquitetura TCP/IP em favor de arquiteturas mais flexíveis que empregam protocolos leves. O principal problema com arquiteturas de comunicação convencionais é que não existe um único protocolo que seja ótimo para todas as aplicações pois cada aplicação tem um conjunto específico de requisitos relacionados à comunicação. Ao invés de projetar um único protocolo que atenda as necessidades de uma ampla gama de aplicações, parece ser mais factível desenvolver um componente que permita a seleção, configuração e combinação de protocolos leves previamente implementados. Esse paradigma oferece diversas vantagens, incluindo a habilidade de criar novos serviços de comunicação sob demanda e permitir que aplicações experimentem diferentes configurações de protocolo de comunicação, coletando métricas para identificar a melhor configuração para as suas necessidades.

1.2 O Sistema de Comunicação Proposto

Nos próximas capítulos discutiremos a arquitetura de um sistema de comunicação especializado e configurável, constituído por um framework meta-programado, responsável por prover mecanismos que permitem selecionar, configurar e combinar protocolos de comunicação de acordo com os requisitos da aplicação, e um núcleo de comunicação sobre o qual os protocolos são projetados. A premissa básica desse sistema de comunicação é que seja possível manter a modularidade dos protocolos leves, aumentando a reusabilidade, e mesmo assim suportar técnicas de implementação de alto desempenho utilizando um mecanismo de composição explícito no lugar do encapsulamento em camadas.

O sistema de comunicação descrito a seguir foi desenvolvido dentro do contexto do projeto SNOW [3], que tem por objetivo criar um ambiente de software orientado à aplicação para suportar computação de alto desempenho sobre agregados de PCs dedicados de forma eficiente. Dois artigos foram publicados ao longo do desenvolvimento do trabalho descrito nessa dissertação. Em [4] a arquitetura do sistema de comunicação é apresentada, juntamente com os detalhes referentes a implementação de um núcleo básico de comunicação para a tecnologia de rede Myrinet. Em [5] a arquitetura do framework meta-programado do sistema de comunicação é descrita em detalhes.

O restante dessa dissertação está dividido como descrito a seguir. O capítulo 2 apresenta uma descrição do sistema de programação paralela SNOW e da área para a qual SNOW foi projetado: computação de alto desempenho sobre agregados de PCs dedicados. Os capítulos 3 e 4 descrevem

as decisões de projeto relacionadas ao sistema de comunicação empregado pelo SNOW e apresentam os detalhes de uma implementação para a tecnologia de rede Myrinet, utilizada como estudo de caso. O capítulo 5 conclui essa dissertação e discorre a respeito de trabalhos futuros.

Capítulo 2

Computação de Alto Desempenho sobre Agregados de PCs

A tecnologia de *miniaturização de componentes eletrônicos* permite que mais transistores sejam empilhados na mesma área de uma placa de silício, o que implica em processadores mais rápidos e memórias com mais capacidade. Apesar de os fabricantes de processadores, guiados pela lei de Moore [6], terem mantido uma impressionante taxa de crescimento no desempenho dos componentes que produzem, existe um limite físico para a miniaturização de componentes eletrônicos.

O *paralelismo*, que em conjunto com a tecnologia de miniaturização é o principal responsável pelo aumento considerável no desempenho dos sistemas computacionais, consiste na divisão do total de processamento a ser efetuado entre um conjunto de unidades básicas com o objetivo de diminuir o tempo total gasto com processamento, aumentando o desempenho do sistema computacional. Ao contrário da tecnologia de miniaturização, não existe um limite físico para o paralelismo, o que vem intensificando a pesquisa em volta do tema e consagrando arquiteturas paralelas como a principal tendência na área de computação de alto desempenho.

Um exemplo de paralelismo e também uma técnica comum utilizada no projeto de unidades lógico-aritméticas em processadores modernos é a utilização de diversos somadores, um para cada bit da palavra na arquitetura, com o objetivo de possibilitar que uma operação de soma possa ser efetuada em um único ciclo de relógio. O *pipelining*, outra técnica que atualmente é consenso no projeto de processadores, emprega o paralelismo em nível de instrução com o objetivo de aumentar o número de instruções executadas a cada unidade de tempo. Além de empregar paralelismo dentro dos processadores, sistemas computacionais modernos comumente utilizam diversos pro-

cessadores trabalhando em paralelo com o objetivo de aumentar o desempenho. De fato, a grande maioria dos sistemas computacionais de alto desempenho é constituída de sistemas paralelos que chegam a dividir o processamento a ser executado entre milhares de processadores. A figura 2.1 apresenta a distribuição dos 500 sistemas computacionais de alto desempenho mais eficientes do mundo de acordo com o número de processadores que esses sistemas utilizam.

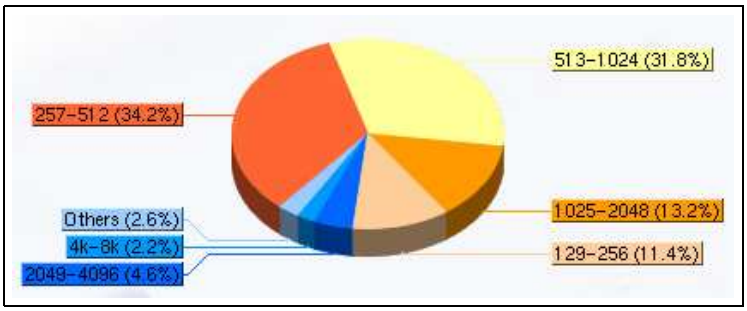

Figura 2.1: Distribuição dos 500 supercomputadores mais eficientes do mundo de acordo com o número de processadores utilizado [2].

2.1 Paradigmas de Programação Paralela

Devido ao trabalho desenvolvido pelos projetistas de hardware, que criaram conjuntos de instruções que escondem o paralelismo empregado nos processadores modernos provendo um modelo de programação seqüencial, e pelos desenvolvedores de compiladores, que têm se esforçado para criar compiladores capazes de re-arranjar o código compilado com o objetivo de aproveitar melhor o paralelismo em nível de instrução, hoje somos capazes de desfrutar do paralelismo embutido nos processadores modernos sem mudar a maneira como escrevemos nossos programas. Por outro lado, para tirar proveito do paralelismo *entre* processadores os engenheiros de software se vêem obrigados a desenvolver código explicitamente paralelo, pois ainda não existem linguagens que nos permitam expressar paralelismo de forma transparente ou compiladores capazes de explorar paralelismo de alta granularidade, empregando múltiplos processadores de forma eficiente.

Com o objetivo de facilitar a vida dos desenvolvedores, diversos paradigmas de programação paralela foram propostos, dentre os quais podemos destacar:

- **Memória compartilhada:** Diversos processos compartilham o mesmo espaço de endereçamento e comunicam-se entre si através de estruturas de dados localizadas na memória compartilhada. Esse modelo de programação provê baixa latência e alta largura de banda na comunicação entre processos e é suportado em hardware por arquiteturas SMP

(*symmetric multiprocessors*). Dentre os projetos que disponibilizam ferramentas para esse paradigma podemos destacar o Shrimp, Posix Threads (Pthreads) e OpenMP, que vêm definindo uma API padrão para o paradigma de memória compartilhada.

- **Troca de mensagens:** No paradigma de troca de mensagens, os processos comunicam-se entre si através do envio e recebimento de mensagens. Implementações desse paradigma, tais como a *Message Passing Interface* (MPI) e a *Parallel Virtual Machine* (PVM), comumente oferecem diversos métodos diferentes para o envio e o recebimento de mensagens, como por exemplo o envio e recebimento ponto-a-ponto e o envio e recebimento multiponto, além de operações coletivas. O MPI é considerado o padrão *de facto* para esse paradigma de programação paralela.

- **Paralelismo de dados:** Empregar o paralelismo de dados consiste em manipular simultaneamente vários subconjuntos de estruturas de dados quaisquer, tais como vetores e matrizes. Esse paradigma de programação paralela é suportado em hardware por supercomputadores vetoriais e pode ser empregado por qualquer tipo de computador paralelo desde que as operações aplicadas a um determinado subconjunto não dependam do resultado das operações aplicadas a outros subconjuntos. Um programa que utiliza o paradigma de paralelismo de dados é essencialmente um programa seqüencial onde o programador ou o compilador determina que estruturas de repetição podem ser executadas em paralelo e como os dados devem ser distribuídos entre os processadores. Paralelismo de dados tem sido implementado com *High-Performance Fortran* (HPF) e *Bulk Synchronous Parallel* (BSP).

- **Objetos compartilhados:** Nesse paradigma, processos podem compartilhar dados desde que esses dados estejam armazenados em objetos e sejam acessados através da invocação de métodos desses objetos. Apesar desse paradigma não ser tão difundido quanto os outros, ele vem recebendo bastante atenção dos pesquisadores da área de programação paralela. Objetos compartilhados são geralmente implementados utilizando linguagens de alto nível como Java.

2.2 Organização da Memória

Um aspecto a ser considerado quando um paradigma de programação é escolhido para um sistema paralelo é a organização da memória do sistema. Sistemas paralelos podem ser divididos, de

acordo com a organização de memória que empregam, em dois tipos: *sistemas de memória compartilhada* e *sistemas de memória distribuída*. Sistemas de memória compartilhada disponibilizam um mesmo espaço de endereçamento de memória para todos os processadores do sistema, e esses processadores têm acesso aos mesmos conjuntos de dados e instruções localizados na memória compartilhada. Nos sistemas de memória distribuída, cada processador tem uma memória local exclusiva que não pode ser acessada por nenhum dos outros processadores do sistema e, conseqüentemente, cada processador tem os seus próprios conjuntos de dados e instruções. O termo *Multicomputador* é utilizado por diversos autores para se referir a esse tipo de sistema, já que cada bloco funcional de um sistema de memória distribuída é um computador completo, com processador e memória. É importante observar que a diferença entre essas duas organizações de memória é caracterizada pela maneira que o subsistema de memória do processador interpreta um endereço de memória. Ou seja, a diferença está na estrutura da memória virtual do sistema. Fisicamente, ambos os sistemas de memória são particionados em componentes que podem ser acessados independentemente.

A organização da memória é extremamente importante pois determina como as diferentes partes de uma aplicação paralela vão se comunicar e influencia diretamente a maneira como o sistema é programado. Em um ambiente de memória compartilhada uma estrutura localizada na memória pode ser utilizada para viabilizar a comunicação entre os processadores do sistema. Além disso, sistemas de memória compartilhada possibilitam que abstrações projetadas para viabilizar a comunicação e sincronização de processos paralelos em sistemas operacionais, tais como semáforos e monitores, sejam adaptadas e utilizadas para coordenar a comunicação entre processadores, tornando a curva de aprendizado relacionada ao desenvolvimento de aplicações paralelas para esse tipo de sistema menos acentuada. Uma das principais desvantagens dessa organização de memória é o problema de coerência de cache. Já que todos os processadores acessam os mesmos dados na memória compartilhada, é necessário empregar mecanismos que invalidem as cópias desses dados na memória cache dos processadores para que não haja o risco de um processador utilizar dados desatualizados [7].

Em um ambiente de memória distribuída, cópias das estruturas de dados compartilhadas devem ser criadas nas memórias locais de cada processador e essas cópias são atualizadas através de trocas de mensagens entre os processadores. Existem diversas vantagens relacionadas a essa organização de memória: primeiro, cada processador pode utilizar toda a banda disponível para acessar a sua memória local, sem interferência de outros processadores; segundo, como não existe um barramento compartilhado entre os processadores do sistema, também não existe um limite

para o número de processadores no sistema - o tamanho do sistema computacional é limitado apenas pela rede utilizada para interconectar os processadores; terceiro, o problema de coerência de cache não existe nesse tipo de sistema pois cada processador acessa apenas a sua memória local. A maior desvantagem desse tipo de sistema paralelo é que a comunicação entre processadores se torna mais difícil, requerendo que mensagens sejam trocadas entre os processadores do sistema toda vez que um processador requer dados de um outro processador. Essa característica introduz duas fontes de atraso: o tempo necessário para construir as mensagens a serem trocadas e o fato de que o processador do nodo receptor deve ser interrompido de maneira a lidar com as mensagens recebidas. Além disso, semáforos, monitores e outras técnicas de programação concorrente não são diretamente aplicáveis em máquinas com memória distribuída. Entretanto, essas técnicas podem ser implementadas através de camadas de software que utilizam troca de mensagens para prover um ambiente de programação similar ao de sistemas de memória compartilhada.

Todos os quatro paradigmas de programação mencionados anteriormente foram implementados nos dois tipos de hardware paralelo. A tabela 2.1 apresenta as diferentes implementações de paradigmas de programação paralela para sistemas de memória compartilhada e distribuída [8].

Paradigma de programação	**Memória compartilhada**	**Memória distribuída**
Memória compartilhada	Pthreads	Shasta, Tempest e TreadMarks
Troca de mensagens	MPI	MPI e PVM
Paralelismo de dados	OpenMP	HPF
Objetos compartilhados	Java	Orca, CRL e Java/RMI

Tabela 2.1: Implementações de paradigmas de programação paralela para sistemas de memória compartilhada e distribuída.

2.3 Sistemas Computacionais de Alto Desempenho

Três tipos de arquitetura dominam o cenário atual da computação de alto desempenho: *multiprocessadores*, *multicomputadores* e *processadores vetoriais*. Sistemas computacionais que utilizam processadores vetoriais, tais como os diversos supercomputadores desenvolvidos ao longo dos anos pela empresa que hoje é conhecida como Cray Inc., podem executar diversas operações matemáticas simultaneamente em estruturas de dados como matrizes e vetores, acelerando a execução de simulações complexas e transformações em gráficos 3D. Supercomputadores baseados em processadores vetoriais foram a arquitetura dominante na área de computação de alto desempenho

entre o final da década de 70 e o início da década de 90. Entretanto, devido à complexidade relacionada ao projeto e implementação de aplicações sobre processadores vetoriais, ao alto custo associado ao desenvolvimento de supercomputadores que utilizam essa arquitetura e ao aumento quase exponencial no desempenho de processadores escalares, computadores vetoriais perderam sua hegemonia para arquiteturas que empregam diversos processadores escalares trabalhando em paralelo. Hoje diversos dos conceitos e técnicas de implementação explorados por processadores vetoriais são utilizados nas unidades de processamento de gráficos, ou *Graphics Processing Units* (GPU), presentes em adaptadores de vídeo modernos. Além disso, processadores de propósito específico como o *Cell*, desenvolvido pela IBM, Toshiba e Sony para ser utilizado na próxima geração de video-games, utilizam processadores vetoriais para acelerar a execução de aplicações em certos domínios.

Multiprocessadores são sistemas computacionais paralelos onde os vários processadores do sistema compartilham um mesmo espaço de endereçamento. Sistemas multiprocessados utilizam ambientes de hardware de memória compartilhada, onde tecnologias de interconexão dedicadas são utilizadas para interligar todos os processadores à memória principal do sistema, ou camadas de software que emulam um espaço de endereçamento compartilhado em um ambiente onde a memória é distribuída entre os processadores do sistema. Arquiteturas SMP, onde dois ou mais processadores conectados à mesma memória principal trabalham em conjunto sob a coordenação do sistema operacional, são o exemplo mais tradicional de multiprocessador e vêm se tornando cada vez mais comuns em PCs modernos. Na arquitetura ccNUMA (do inglês *Cache Coherent Non-Uniform Memory Access*) a memória física é distribuída, permitindo que cada processador possa acessar a sua própria memória de maneira eficiente, mas a memória lógica é compartilhada, possibilitando que estruturas de dados sejam compartilhadas entre os diversos processadores do sistema e que um processador possa acessar a memória de outros processadores. Mecanismos de coerência de cache, implementados em hardware ou software, são utilizados para garantir que as estruturas de dados compartilhadas pelos processadores estejam sempre atualizadas.

Multicomputadores são sistemas paralelos de memória distribuída compostos de uma coleção de nodos de processamento interligados por uma rede de comunicação de alta velocidade, onde cada um dos nodos de processamento é um sistema completo, com memória e processador, capaz de operar independentemente dos outros. Dentre as diferentes arquiteturas de multicomputador podemos destacar os *processadores massivamente paralelos* (MPPs) e os *agregados de PCs*. MPPs são sistema computacionais que empregam processadores, rede de interconexão e ambientes de software especializados. Em contrapartida, agregados são geralmente montados utilizando-se

ambientes de software, nodos de processamentos e redes de comunicação disponíveis comercialmente, o que torna o sistema computacional mais acessível e permite que as inovações tecnológicas implementadas pelos fabricantes de processadores e redes de comunicação sejam rapidamente incorporadas. Apesar de a maioria dos agregados de PCs modernos utilizarem processadores SMP nos seus nodos de processamento, agregados são considerados sistemas computacionais de memória distribuída pois a quantidade de nodos em um agregado é muito maior do que a quantidade de processadores em um nodo.

Assim como os agregados de PCs, constelações são arquiteturas paralelas que utilizam tanto o paradigma de memória distribuída quanto o paradigma de memória compartilhada. Entretanto, devido ao fato de disponibilizarem mais processadores em um único nodo do que o total de nodos que compõe o sistema computacional, constelações geralmente são classificadas como sistemas de memória compartilhada. A distinção entre agregados e constelações, apesar de muitas vezes ignorada, é importante e pode ter um sério impacto na maneira que o sistema é programado. Por exemplo, é provável que um agregado seja programado quase que exclusivamente com um sistema de troca de mensagens, como por exemplo o MPI, enquanto uma constelação provavelmente seria programada, pelo menos em parte, com OpenMP utilizando um modelo com várias *threads* de processamento.

É importante observar que não existe um padrão oficial para as terminologias utilizadas para designar as diferentes arquiteturas de sistemas computacionais de alto desempenho [9]. De fato, os termos MPP, agregados de PCs e constelações são muitas vezes utilizados como sinônimos apesar das diferenças gritantes entre essas arquiteturas [10]. A figura 2.2 apresenta a distribuição dos 500 supercomputadores mais eficientes do mundo em 2005 de acordo com a sua arquitetura. A terminologia apresentada no gráfico foi definida pelos autores do levantamento [2].

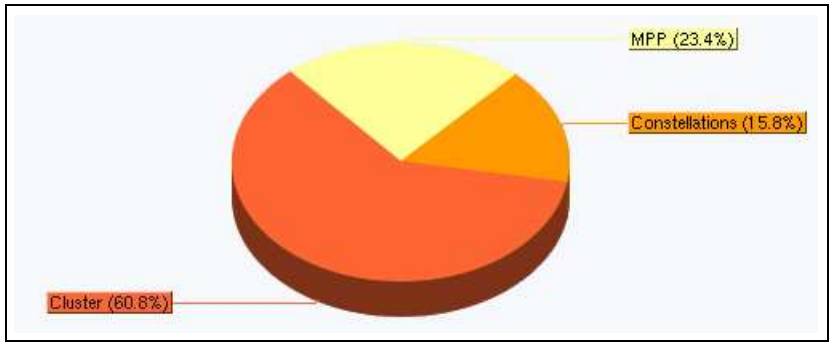

Figura 2.2: Distribuição dos 500 supercomputadores mais eficientes do mundo de acordo com a sua arquitetura [2].

2.3.1 Ambientes de Software Especializados para Agregados de PCs

Os primeiros agregados de PCs, que surgiram há cerca de dez anos, utilizavam o conceito de empregar componentes pré-fabricados tanto para o ambiente de hardware quanto para o ambiente de software do sistema computacional. Essa classe de agregados recebeu a denominação de Beowulf e popularizou a adoção do sistema operacional de propósito geral Linux como o sistema de suporte em tempo de execução utilizado pelos nodos de processamento dos agregados de PCs. Agregados de PCs Linux ainda hoje desempenham um importante papel na área de computação de alto desempenho e no setor comercial, sendo considerados o sistema computacional paralelo mais amplamente difundido.

O problema com a estratégia de utilizar software de propósito geral em computação sobre agregados é que os sistemas operacionais de propósito geral, tais como o Unix/Linux e o Windows, foram projetados sem levar em consideração os requisitos específicos das aplicações e do ambiente de hardware paralelos, o que afeta o desempenho, a usabilidade, a escalabilidade e a adaptabilidade do sistema computacional. O grande número de projetos de pesquisa na área de computação paralela sobre agregados de PCs que propõe camadas *middleware* sobre o sistema operacional, destacando-se entre eles as implementações de soluções para troca de mensagens (*message passing*) [11, 12, 13] e para simulação de ambientes de memória compartilhada distribuída (*distributed shared memory*) [14], serve como fundamento à suposição de que ambientes de software comuns não são adequados para suportar computação de alto desempenho sobre agregados. Se os sistemas operacionais de propósito geral utilizados pela maioria dos agregados em funcionamento fossem capazes de atender as necessidades das aplicações paralelas, ou ao menos disponibilizassem arquiteturas extensíveis que possibilitassem a adição dos serviços requeridos por essas aplicações de forma eficiente, muitos desses projetos não seriam necessários.

Apesar da maioria dos grupos que realizam computação sobre agregados ainda insistir em utilizar software de propósito geral existem diversos projetos de pesquisa que tem por objetivo prover ambientes de software específicos para agregados de PCs [15, 16]. A utilização de ambientes de software especializados na área de computação de alto desempenho sobre agregados de PCs, especialmente no que se refere aos sistemas operacionais utilizados pelos nodos de processamento dos agregados, permite que os recursos de hardware sejam utilizados mais eficientemente. Sistemas operacionais de propósito geral geralmente implementam em seu núcleo serviços que são imprescindíveis em um computador pessoal, tais como escalonamento, gerenciamento complexo da memória do sistema e multiplexação/proteção para os componentes de hardware, mas que não

fazem sentido em um ambiente especializado como os nodos de processamento de um agregado de PC. Esses serviços poderiam tranquilamente ser deixados de fora do sistema operacional aumentando o desempenho dos nodos de processamento do agregado e conseqüentemente contribuindo com um aumento no desempenho final das aplicação paralelas. Projetos de pesquisa tais como o PEACE [17] e o CHOICES [18] utilizam sistemas operacionais especializados que adaptam-se aos requisitos das aplicações e ambiente de hardware paralelos com o objetivo de aumentar o desempenho final do sistema computacional.

2.3.2 Sistemas de Comunicação Especializados para Agregados de PC

Um dos principais problemas relacionados à utilização de sistemas operacionais de propósito geral no domínio de computação de alto desempenho sobre agregados de PCs está relacionado com os sistemas de comunicação disponibilizados por esses sistemas operacionais. Esses sistemas de comunicação não foram projetados de maneira a tirar proveito das recentes inovações tecnológicas implementadas pelos fabricantes de redes, responsáveis pela transição do gargalo relacionado ao desempenho de comunicação do ambiente de hardware para o ambiente de software. Sistemas de comunicação tradicionais, como por exemplo o Internet Sockets/TCP, colocam todo o processamento relacionado aos serviços de comunicação no núcleo do sistema operacional. Como resultado, o caminho crítico durante o envio e recebimento de mensagens inclui operações caras, tais como trocas de contexto, desencadeadas por chamadas de sistemas, e tratamento de interrupções, além de um excessivo número de cópias.

Devido ao fato de que o desempenho de um agregado de PCs depende de mecanismos eficientes de comunicação entre seus nodos de processamento, cientistas da computação tiveram que tirar o sistema operacional do caminho com o objetivo de aumentar o desempenho da comunicação em agregados que utilizam ambientes de software de propósito geral. Os sistemas de comunicação em nível de usuário (*User-Level Communication* ou ULC) [19, 20, 21] são amplamente empregados na área de computação de alto desempenho sobre agregados de PCs, o que pode ser considerado a mais significativa evidência de que a utilização de software de propósito geral não é tão efetiva quanto a utilização de hardware de propósito geral nessa área.

Sistemas ULC são sistemas de comunicação que permitem que a interface de rede seja acessada diretamente pela aplicação, sem a intervenção do sistema operacional. Tecnologias de rede modernas provêm taxas de transmissão que chegam a ordem dos gigabits por segundo e latência na ordem dos microsegundos, o que faz com que as chamadas de sistema utilizadas por sistemas

de comunicação tradicionais no envio e recebimento de mensagens sejam mais dispendiosas do que a transferência de mensagens em si. Sistemas ULC geralmente utilizam o sistema operacional apenas na etapa de inicialização do sistema, onde a interface de rede é inicializada e mapeada no espaço de endereçamento da aplicação e onde *buffers* especiais utilizados durante a comunicação são alocados no espaço de endereçamento do sistema operacional. Além disso, algums sistemas ULC utilizam o sistema operacional durante o processo de comunicação para efetuar tarefas como tradução de endereços lógicos para endereços físicos e o tratamento de interrupções, utilizadas por alguns sistemas ULC para lidar com mensagens recebidas pela rede.

Sistemas ULC geralmente implementam um ou mais protocolos leves sobre o hardware de comunicação no lugar da clássica arquitetura em camadas que guiou o desenvolvimento de protocolos bastante difundidos como o Protocolo Internet (TCP/IP). O principal problema com arquiteturas de comunicação convencionais é que não existe um único protocolo que seja ótimo para todas as aplicações, pois cada aplicação tem um conjunto específico de requisitos relacionados à comunicação. Os sistemas ULC permitem que as aplicações acessem protocolos leves diretamente mas na maioria dos casos camadas *middleware* que encapsulam esses protocolos e provêm interfaces padrão, tais como PVM e MPI, são utilizadas.

A principal vantagem relacionada à utilização de protocolos leves é a flexibilidade e adaptabilidade que um sistema de comunicação baseado nesse tipo de protocolo apresenta [22]. Ao invés de projetar um conjunto fixo de protocolos que atenda as necessidades de uma ampla gama de aplicações, componentes que permitem selecionar, configurar e combinar protocolos leves previamente implementados são utilizados. Esse paradigma oferece diversas vantagens, incluindo a habilidade de criar novos serviços de comunicação sob demanda e permitir que aplicações experimentem com diferentes configurações de protocolo de comunicação, coletando métricas para identificar a melhor configuração para as suas necessidades.

2.3.3 O Projeto SNOW

O desenvolvimento de aplicações paralelas não é uma tarefa trivial. Além da complexidade referente ao projeto e implementação de qualquer aplicação, o desenvolvedor de aplicações paralelas tem que lidar com fatores inerentes aos ambientes paralelos, tais como sincronização e a maior susceptibilidade a falhas relacionadas ao hardware. *Ambientes de programação paralela* suportam ao menos um dos paradigmas de programação paralela existentes e têm como objetivo facilitar a implementação, depuração e execução de aplicações paralelas. Esses ambientes normalmente

disponibilizam linguagens de programação paralela, bibliotecas de funções, depuradores, analisadores de desempenho e ferramentas para gerenciar os recursos do sistema computacional, que provêm soluções para o escalonamento, a alocação e o monitoramento de recursos de hardware e software, tais como tempo de processamento, espaço de armazenamento e canais de comunicação.

Devido à popularização dos agregados de PCs no cenário de computação de alto desempenho, diversos grupos de pesquisa têm se dedicado a projetar ambientes de programação paralela para essa plataforma. O fato de que a maioria dos agregados de PCs disponíveis atualmente utiliza um sistema operacional de propósito geral faz com que a maioria dos ambientes de programação projetados para esse tipo de arquitetura paralela sejam implementados como camadas *middleware* entre o sistema operacional e as aplicações. A principal vantagem em utilizar camadas *middleware* é que as funcionalidades projetadas especificamente para aplicações paralelas são mantidas isoladas do sistema operacional, facilitando a portabilidade dessas implementações entre diversos sistemas operacionais. Entretanto, camadas *middleware* são responsáveis por um impacto indesejável no desempenho dos agregados, o que tem motivado diversos grupos de pesquisa a propor ambientes de programação paralela onde até mesmo o sistema operacional é projetado conforme os requisitos das aplicações e hardware paralelos.

O projeto SNOW, desenvolvido em parceria pela Universidade Federal de Santa Catarina (UFSC), Universidade Federal do Rio Grande do Sul (UFRGS) e o instituto de pesquisa Fraunhofer-FIRST na Alemanha, tem por objetivo desenvolver um ambiente de programação que possa suportar computação paralela em agregados de PCs dedicados de forma eficiente. Este ambiente inclui um sistema operacional especializado e minimalista, contendo apenas as abstrações e serviços necessários para atender aos requisitos da aplicação paralela em questão, que é gerado especificamente para cada aplicação de acordo com os seus requisitos. Além disso, o sistema também disponibiliza uma linguagem de programação paralela baseada em C++ e ferramentas visuais de gerenciamento para os recursos de hardware e software disponíveis no sistema computacional. Com o intuito de permitir que as aplicações existentes sejam executadas no ambiente proposto, o SNOW foi projetado de maneira a ser compatível com as interfaces padrão adotadas pela comunidade de computação paralela, como POSIX e MPI. Além disso, o ambiente de programação paralela SNOW provê um sistema de comunicação configurável e extensível, projetado de maneira a permitir que as aplicações paralelas possam contar com um ambiente de software de comunicação especializado. As próximas seções descrevem os componentes que fazem parte do ambiente de programação paralela SNOW.

2.3.3.1 O Sistema Operacional EPOS

Sistemas operacionais configuráveis são um tema recorrente entre os pesquisadores de sistemas operacionais [23]. O EPOS é um sistema operacional baseado em componentes que provê um conjunto de mecanismos meta-programados e ferramentas de configuração estática capazes de gerar automaticamente um sistema operacional especializado para determinadas classes de aplicações. O sistema operacional EPOS foi desenvolvido de acordo com a metodologia de *Projeto de Sistemas Orientado à Aplicação* (AOSD) [24], uma metodologia que define diretivas para a implementação de sistemas operacionais orientados à aplicação. O sistema foi utilizado dentro do escopo do projeto SNOW com o objetivo de prover às aplicações paralelas um ambiente de suporte em tempo de execução minimalista e de alto desempenho.

O sistema operacional EPOS, acrônimo para *Embedded Parallel Operating System* ou Sistema Operacional Embarcado e Paralelo, foi desenvolvido com o objetivo de embutir o sistema operacional em aplicações paralelas, disponibilizando uma camada de abstração específica para o hardware em uso e o conjunto de serviços necessários para a execução da aplicação. O EPOS consiste em um conjunto de componentes de software, fruto de uma extensa e detalhada análise do domínio de computação de alto desempenho, e um conjunto de ferramentas que permitem que esses componentes sejam combinados e configurados. A figura 2.3 apresenta diagrama relativo ao processo de análise de domínio proposto pela metodologia AOSD, utilizado para gerar os componentes de software que constituem o sistema operacional EPOS.

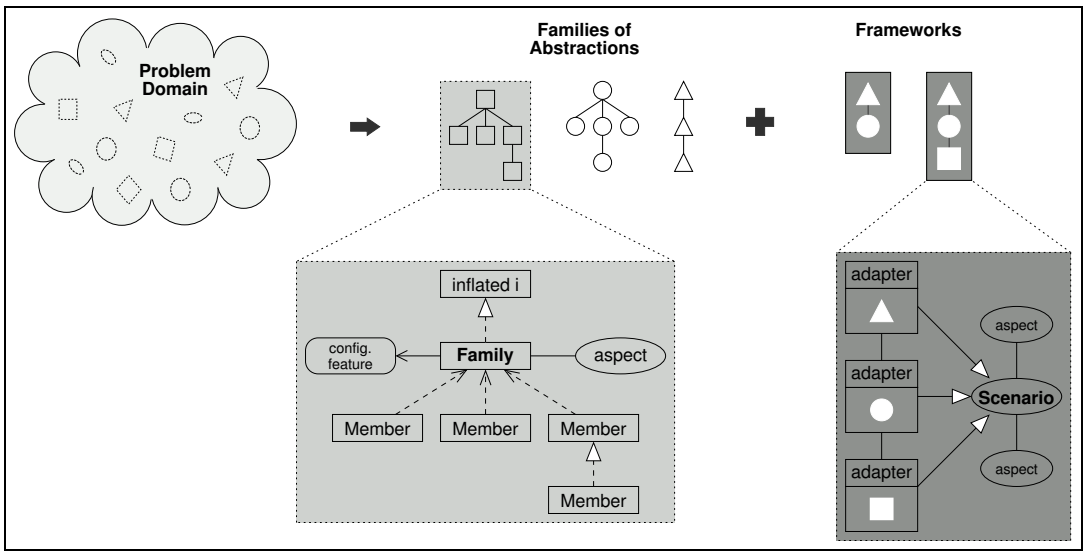

Figura 2.3: Processo de análise de domínio proposto pela metodologia AOSD [24].

Os componentes que constituem o EPOS são chamados de `Abstrações Independentes`

de Cenário e consistem em classes que implementam abstrações plausíveis em um sistema de suporte em tempo de execução qualquer, tais como escalonamento, comunicação entre processos e sistema de arquivos. A principal característica das Abstrações Independentes de Cenário é que os fatores relacionados a cenários específicos de aplicação para esses componentes são abstraídos dos componentes e fatorados com técnicas de Programação Orientada a Aspectos. Essa característica promove a reusabilidade desses componentes pois a mesma abstração pode ser utilizada em uma variedade de cenários diferentes.

No EPOS, características de determinados cenários de utilização do sistema operacional são projetados como aspectos que são aplicados às Abstrações Independentes de Cenário. Adaptadores de Cenário são agentes que encapsulam as Abstrações Independentes de Cenário de maneira a intermediar suas interações com componentes dependentes de cenário. Adaptadores de Cenário funcionam de maneira similar aos *weavers* da AOP, mas ao contrário de uma ferramenta separada ou um pré-compilador, são implementados através da utilização de *template metaprogramming*, o suporte que a linguagem C++ disponibiliza para metaprogramação.

Configuração é um fator muito importante no projeto do EPOS pois o principal objetivo do sistema operacional é prover um ambiente de suporte em tempo de execução especializado para cada aplicação. EPOS utiliza um Repositório de Configuração com informações sobre cada uma das Abstrações Independentes de Cenário do sistema, através de classes *Trait Template*. Cada um desses repositórios contém o conjunto de Características Configuráveis disponíveis para a abstração correspondente.

Abstrações Independentes de Cenário são organizadas em famílias que disponibilizam Interfaces Infladas: interfaces hipotéticas que agregam o conjunto de métodos disponibilizados por uma família de abstrações. Aplicações são projetadas de acordo com as Interfaces Infladas disponibilizadas pelo sistema e submetidas à ferramenta Analyzer, que determina quais famílias de abstrações foram utilizadas pela aplicação. A seleção e configuração dos membros específicos de cada família a serem utilizados é delegada ao Configurator, ferramenta automática de configuração que leva em consideração as funcionalidades disponibilizadas por cada membro da família, seu custo e também as informações contidas no Repositório de Configuração com o objetivo de escolher o membro que melhor satisfaz os requisitos da aplicação no cenário atual. A ferramenta Generator trabalha em conjunto com o compilador para gerar uma nova instância de sistema operacional específico para a aplicação em questão. A figura 2.4 apresenta o processo de geração do EPOS.

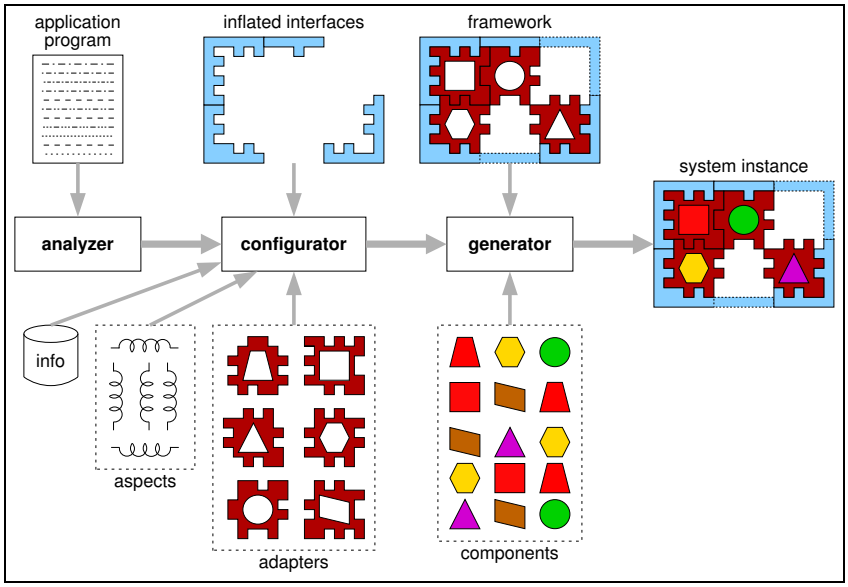

Figura 2.4: Processo de geração do EPOS [24].

A heterogeneidade dos ambientes de hardware nos quais o EPOS pode ser utilizado é abstraída através dos `Mediadores de Hardware` [25], componentes projetados de acordo com um *contrato de interface* pré-estabelecido entre o sistema operacional e os componentes de hardware. Assim como as `Abstrações Independentes de Cenário`, os `Mediadores de Hardware` também são organizados em famílias e selecionados pelas ferramentas de configuração automática do sistema operacional de acordo com as abstrações utilizadas pela aplicação.

A arquitetura em componentes do EPOS associada aos mecanismos automatizados de configuração e geração disponibilizados pelo sistema permite que instâncias especializadas de sistemas operacionais sejam geradas, baseado nos requisitos da aplicação a ser suportada. Um sistema operacional minimalista é gerado e todas as abstrações que não são utilizadas pela aplicação são deixadas de fora, contribuindo com o desempenho e diminuindo o tamanho do sistema.

2.3.3.2 Sistema de Comunicação

Um dos principais objetivos do projeto SNOW foi o estudo de soluções inovativas para o projeto e implementação de sistemas de comunicação para agregados de PCs. De maneira a permitir que instâncias especializadas do sistema de comunicação sejam utilizadas para atender especificamente os requisitos da aplicação em questão e com o objetivo de facilitar a integração com o sistema operacional EPOS, o sistema de comunicação desenvolvido para o SNOW foi projetado de acordo com as diretivas da metodologia de *Projeto de Sistemas Orientado à Aplicação*. Nos

próximos capítulos as decisões de projeto e implementação referentes ao sistema de comunicação desenvolvido para o SNOW serão discutidas. O capítulo 3 descreve em detalhes a arquitetura do sistema de comunicação e o capítulo 4 apresenta uma implementação desse sistema para a tecnologia de rede Myrinet.

2.3.3.3 Interfaces Padronizadas

A maioria dos agregados de PCs disponíveis atualmente utiliza redes Ethernet para interconectar os seus nodos de processamento. Entretanto, a utilização de tecnologias de rede mais modernas, com menor latência e maior largura de banda (Myrinet, SCI, QsNet e Infiniband, entre outras) tem se tornado cada vez mais comum. Embora todas as tecnologias de rede disponibilizem funcionalidades similares, tais como o envio e recebimento de mensagens ponto-a-ponto, cada uma delas possue as suas próprias peculiaridades e disponibiliza interfaces de programação específicas. Padrões como o MPI, padrão *de facto* para o paradigma de programação paralela Troca de Mensagens, permitem que aplicações paralelas possam ser utilizadas sobre diferentes tecnologias de rede, desde que haja uma implementação do padrão para a tecnologia de rede alvo.

As diferenças entre a API que os sistemas operacionais disponibilizam para os programadores é um problema que não está restrito ao cenário de computação de alto desempenho sobre agregados. Com o objetivo de solucionar esse problema o instituto IEEE definiu o POSIX, uma série de padrões utilizados por desenvolvedores de sistemas operacionais e aplicações com o objetivo de promover a portabilidade. O POSIX (do inglês *Portable Operating System Interface* ou interface portável para sistemas operacionais) define interfaces padronizadas para diversos componentes de um sistema operacional, tais como *threads*, *shell* e I/O.

De maneira a permitir que as aplicações paralelas existentes possam ser suportadas pelo SNOW, as interfaces padrão adotadas pela comunidade de computação paralela são suportadas pelo sistema. O EPOS suporta o padrão POSIX nativamente e uma implementação de MPI para o EPOS [26] foi desenvolvida dentro do contexto do projeto SNOW. O padrão MPI foi implementado como um conjunto de aspectos de cenários que leva o sistema de comunicação do EPOS a se comportar de acordo com os serviços previstos no padrão.

Suporte a outros padrões utilizados pela comunidade envolvida com computação de alto desempenho, tais como o PVM e o VIA, deverá ser desenvolvido no futuro para que uma quantidade maior de aplicações paralelas possa ser suportada pelo SNOW.

2.3.3.4 Ferramentas de Gerenciamento

Apesar dos esforços de diversos grupos de pesquisas, o desenvolvimento de ferramentas de gerenciamento para os recursos de hardware e software de agregados de PCs ainda está em sua infância e um padrão para esse tipo de ferramenta ainda não emergiu [27]. O SNOW utiliza o sistema CODINE [28] associado a outras ferramentas desenvolvidas dentro do contexto do projeto de maneira a prover soluções para a alocação e o monitoramento de recursos de hardware e software do sistema computacional.

O CODINE (do inglês *COmputing in DIstributed Networked Environments* ou computação em ambientes distribuídos interconectados) é um sistema de gerenciamento de tarefas que permite que os processadores que constituem um sistema computacional paralelo possam ser tratados como uma única máquina, para a qual o usuário pode submeter tarefas. O CODINE distribui essas tarefas de forma balanceada entre os recursos do sistema computacional. O sistema pode ser controlado através de uma interface gráfica ou linha de comando.

2.3.3.5 Linguagem de Programação Paralela

A linguagem de programação paralela adotada pelo SNOW é o DPC++ (*Data Parallel C++*) [29], uma linguagem orientada a objetos baseada em C++ desenvolvida com o objetivo de suportar programação distribuída. A DPC++ propõe um modelo de programação distribuído adequado ao paradigma de orientação a objetos, proporcionando, através da exploração eficiente do paralelismo de dados, um ambiente de desenvolvimento que une os recursos da programação orientada a objetos aos benefícios do processamento distribuído.

Capítulo 3

O Sistema de Comunicação Proposto

A utilização de ambientes de programação paralela geralmente implica em um aumento considerável no tráfego de mensagens do sistema computacional. Além dos dados a serem processados e dos resultados referentes ao processamento desses dados, o ambiente de programação paralela necessita que mensagens de controle sejam trocadas com os nodos de processamento com o objetivo de avaliar fatores como a disponibilidade de recursos de hardware e o andamento das tarefas de processamento. Como conseqüência, o desempenho de um ambiente de programação paralela está intimamente relacionado à eficiência dos mecanismos de comunicação empregados pelo sistema computacional, especialmente no que se refere aos ambientes de programação para agregados de PCs.

Um dos fatores que tem um impacto considerável no desempenho de sistemas de comunicação tradicionais é a sua natureza monolítica. Com o objetivo de esconder a complexidade das diferentes redes físicas, sistemas de comunicação tradicionais geralmente provêm aos seus usuários um conjunto fixo de serviços e uma interface fixa de acesso a esses serviços. Embora uma interface fixa de acesso aos serviços de um sistema de comunicação seja um restrição necessária e até mesmo benéfica, diferentes aplicações necessitam de diferentes combinações de serviços de comunicação. Para algumas aplicações, o conjunto de serviços providos por um determinado sistema de comunicação pode ser insuficiente, problema comum que geralmente é contornado através da utilização de camadas *middleware* que disponibilizam os serviços necessários mas degradam o desempenho do sistema.

De fato, dois dos principais requisitos referentes ao projeto de sistemas de comunicação modernos são a extensibilidade e a configurabilidade. Sistemas de comunicação extensíveis permitem que novos serviços requisitados por uma determinada aplicação sejam implementados de maneira

eficiente diretamente dentro do sistema de comunicação, dispensando a utilização de camadas *middleware*. A configurabilidade pode melhorar o desempenho da comunicação permitindo que serviços não utilizados sejam desligados e outros aspectos relacionados à comunicação, tais como unidade máxima de transmissão (MTU), tamanho dos *buffers* utilizados durante o envio e recebimento de mensagens e até mesmo o algoritmo de comunicação em si, sejam adaptados de acordo com requisitos particulares.

O sistema de comunicação proposto para o ambiente de programação paralela SNOW é um sistema extensível e configurável projetado especificamente para o domínio de computação de alto desempenho sobre agregados computacionais dedicados. Com o objetivo de prover software de comunicação especializado e de alto desempenho, esse sistema de comunicação associa técnicas de projeto e implementação consagradas por diversos sistemas ULC a um suporte meta-programado responsável por gerar configurações de protocolo especializadas de acordo com os requisitos das aplicações e as características da rede [5]. Ao invés de implementar um protocolo de comunicação monolítico o sistema de comunicação discutido nesse capítulo disponibiliza um conjunto de protocolos leves que podem ser utilizados, individualmente ou em conjunto, para atender aos requisitos específicos das aplicações e permite que novos protocolos sejam facilmente criados e adicionados ao sistema sob demanda.

3.1 Arquitetura do Sistema de Comunicação

A arquitetura do sistema de comunicação proposto para o ambiente de programação paralela SNOW possibilita que implementações especializadas, e geralmente mais eficientes, sejam utilizadas para interagir com o hardware de rede enquanto serviços de comunicação podem ser implementados de maneira genérica. As entidades que compõem o sistema são divididas em três tipos:

Núcleo Básico: Uma abstração que implementa um algoritmo de comunicação minimalista para uma determinada tecnologia de rede. O `Núcleo Básico` define `Pontos de Ação` onde as `Estratégias de Comunicação` ativas são invocadas em tempo de execução com o objetivo de prover os serviços de comunicação que elas implementam. `Núcleos Básicos` são dependentes da tecnologia de rede em uso, pois cada tecnologia de rede possui recursos específicos que podem ser utilizados para aumentar o desempenho da comunicação.

Estratégias de Comunicação: Componentes auto-contidos que implementam desde pequenas

modificações no algoritmo de comunicação até protocolos leves completos. Estratégias de Comunicação podem ser *genéricas*, isto é, independentes da tecnologia de rede física sendo utilizada, ou *especializadas*, projetadas especificamente para uma determinada tecnologia de rede. Estratégias de Comunicação genéricas podem combinar-se com qualquer Núcleo Básico do sistema e Estratégias especializadas combinam-se apenas com o Núcleo Básico correspondente.

Framework Meta-Programado: Responsável por disponibilizar mecanismos que permitem selecionar, configurar e combinar diferentes Estratégias de Comunicação em tempo de compilação e por suportar as interações entre o Núcleo Básico em uso e as Estratégias de Comunicação ativas. Além disso, o Framework Meta-Programado também implementa o repositório de configuração estática para o sistema de comunicação.

As próximas seções descrevem em detalhes as características dessas entidades e a figura 3.1 ilustra a interação entre elas. A interação entre os Núcleos Básicos e as Estratégias de Comunicação foi inspirada nas idéias de *fatoração de aspectos ortogonais ao domínio* popularizada pela Programação Orientada a Aspectos (AOP), onde abstrações podem agir em pontos pré-definidos de uma determinada classe com o objetivo de adicionar funcionalidades ou mudar o seu comportamento. Os Pontos de Ação definidos pelos Núcleos Básicos em seus algoritmos, que servem de interface entre as Estratégias de Comunicação e os Núcleos Básicos, podem ser comparados com *pointcuts*, os pontos definidos para a aplicação de aspectos em uma determinada classe na AOP.

3.1.1 Núcleo Básico

Um Núcleo Básico de comunicação consiste em uma implementação minimalista de um algoritmo de comunicação para uma determinada tecnologia de rede física. O principal objetivo do Núcleo Básico é suportar os protocolos leves implementados pelas Estratégias de Comunicação, contribuindo com a modularidade e portabilidade do sistema e conseqüentemente facilitando o processo de combinação das Estratégias de Comunicação. De fato, se cada uma dessas Estratégias re-implementasse todo o algoritmo de comunicação seria impossível combiná-las sem aumentar a complexidade, e possivelmente diminuir o desempenho, do sistema como um todo.

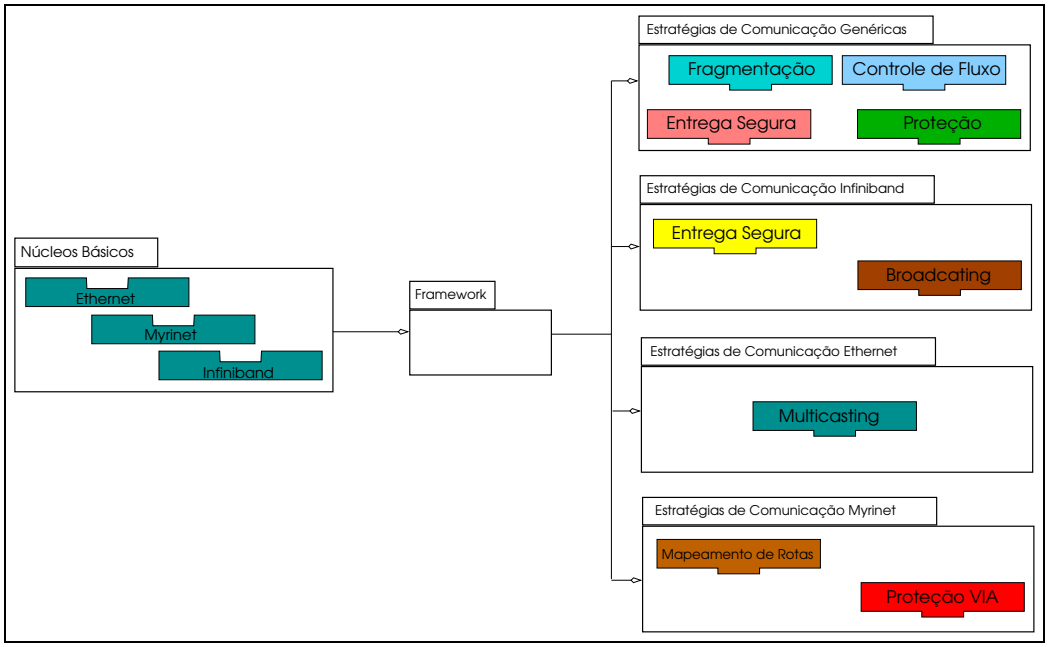

Figura 3.1: Interação entre as entidades que compõe o sistema de comunicação.

De maneira a facilitar o desenvolvimento das `Estratégias de Comunicação`, o `Núcleo Básico` projetado para uma determinada tecnologia de rede deve ser extremamente simples e altamente configurável. Só o que é estritamente necessário para efetuar trocas de mensagens entre dois nodos da rede é implementado no `Núcleo Básico` e até mesmo serviços básicos como fragmentação devem ser projetados como `Estratégias de Comunicação` para que aplicações que não necessitam desses serviços possam desligá-los. Um `Núcleo Básico` deve ser configurável pois determinadas `Estratégias de Comunicação` vão alterar características do seu algoritmo com o objetivo de prover os serviços que implementam. Além disso, um `Núcleo Básico` de comunicação configurável permite que as aplicações possam definir fatores como MTU e tamanho dos *buffers* utilizados durante o envio e recebimento de mensagens.

Com o objetivo de prover às aplicações um sistema de comunicação de alto desempenho, o `Núcleo Básico` deve ser o mais otimizado possível. Essa é a principal motivação por trás da correlação de um-para-um entre os `Núcleos Básicos` de comunicação disponíveis e as tecnologias de rede suportadas pelo sistema de comunicação. Para cada tecnologia de rede suportada um `Núcleo Básico` de comunicação deve ser desenvolvido, utilizando as funcionalidades disponibilizadas especificamente por essa tecnologia de rede para extrair o máximo de desempenho do hardware de comunicação.

Uma outra responsabilidade importante do `Núcleo Básico` de comunicação é definir a interface de acesso ao sistema de comunicação. Apesar de o sistema de comunicação ser confi-

gurável, sua interface de acesso deve permanecer fixa para facilitar a utilização e a interoperabilidade do sistema. Além disso, o Núcleo Básico é responsável por definir Pontos de Ação, pontos no algoritmo de comunicação onde os protocolos implementados pelas Estratégias de Comunicação vão poder agir de maneira a prover os seus serviços. É importante observar que os Pontos de Ação, assim como a implementação do Núcleo Básico, vão variar de acordo com a tecnologia de rede utilizada. Entretanto, Pontos de Ação relativamente genéricos podem ser propostos:

Inicialização: Permite que as Estratégias de Comunicação ativas possam configurar-se durante a etapa de inicialização do Núcleo Básico.

Antes do envio: Agindo nesse ponto, uma determinada Estratégia pode por exemplo re-implementar o algoritmo de envio da arquitetura básica de comunicação.

Permitir envio: Esse Ponto de Ação pode ser utilizado para inviabilizar o envio de uma mensagem, caso exista algum problema com a criação do quadro (*frame*) por exemplo.

Após o envio: Uma Estratégia de Comunicação que provê entrega segura pode agir nesse ponto com o objetivo de esperar por uma confirmação do receptor e ativar o mecanismo de re-envio caso a confirmação não chegue.

Antes do recebimento: Pode ser utilizado para re-implementar o algoritmo de recebimento da arquitetura básica ou para garantir que haja espaço nos *buffers* utilizados para o recebimento de mensagens antes de iniciar o recebimento em um protocolo de controle de fluxo.

Permitir recebimento: Utilizado para inviabilizar o recebimento de mensagens caso exista algum problema, tal como espaço insuficiente nos *buffers* de recebimento de mensagens.

Após o recebimento: Na implementação do serviço de entrega segura, esse Ponto de Ação seria utilizado pela Estratégia de Comunicação para enviar uma confirmação de recebimento.

Finalização: Permite que as Estratégias de Comunicação preparem-se para finalizar sua execução, liberando toda a memória alocada por exemplo.

Conforme mencionado anteriormente, a interação entre Núcleos Básicos e Estratégias de Comunicação foi inspirada em conceitos utilizados na Programação Orientada a Aspectos. De fato, os Pontos de Ação definidos pelos Núcleos Básicos em

seus algoritmos são bastante similares aos *pointcuts* utilizados por aspectos na AOP para interagir com a classe alvo. A principal diferença entre os dois conceitos é o fato de os *pointcuts* da AOP serem definidos através da utilização de uma linguagem especializada na própria declaração do aspecto, de forma transparente para a classe na qual os aspectos vão agir, enquanto os `Pontos de Ação` são definidos explicitamente pelos `Núcleos Básicos` nos pontos de configuração presentes em seus algoritmos de comunicação. Além disso, na maioria das linguagens que suportam aspectos os *pointcuts* só podem ser definidos antes, depois ou ao redor (antes e depois) de métodos de uma determinada classe. Os `Pontos de Ação` por outro lado podem ser declarados por um determinado `Núcleo Básico` em qualquer parte de seu algoritmo de comunicação.

A figura 3.2 apresenta o fluxograma correspondente ao algoritmo de comunicação de um `Núcleo Básico` hipotético. O `Núcleo Básico` passa por uma etapa de inicialização e em seguida fica em um estado de espera até que requisições de envio ou recebimento de quadros sejam efetuadas ou até que o sistema seja finalizado. O algoritmo de envio consiste em montar um quadro com as informações disponíveis e em seguida copiá-lo para a interface de rede, que se encarrega de enviá-lo. No recebimento, as informações presentes no cabeçalho do quadro são analisadas e, dependendo dos valores dos campos do cabeçalho, o quadro é copiado para o *buffer* da aplicação. Círculos são utilizados no fluxograma para representar os `Pontos de Ação` definidos por esse `Núcleo Básico` em determinados pontos de seu algoritmo de comunicação. Além disso, os `Pontos de Ação` genéricos discutidos anteriormente estão indicados na figura.

3.1.2 Estratégias de Comunicação

Como mencionado anteriormente, `Estratégias de Comunicação` são componentes auto-contidos que implementam desde pequenas modificações no algoritmo de comunicação provido pelo `Núcleo Básico` em uso até protocolos leves completos. O termo auto-contido é utilizado para indicar que uma determinada `Estratégias de Comunicação` é responsável pela implementação completa de um determinado serviço de comunicação, definindo os algoritmos a serem utilizados tanto no envio quanto no recebimento de quadros.

O serviço implementado por uma determinada `Estratégia de Comunicação` deve ser projetado de acordo com os `Pontos de Ação` que os `Núcleos Básicos` provêm. A decomposição de serviços de comunicação complexos em `Estratégias` exige mais esforço do que um projeto monolítico, entretanto, conforme demonstrado pelas diversas aplicações e tra-

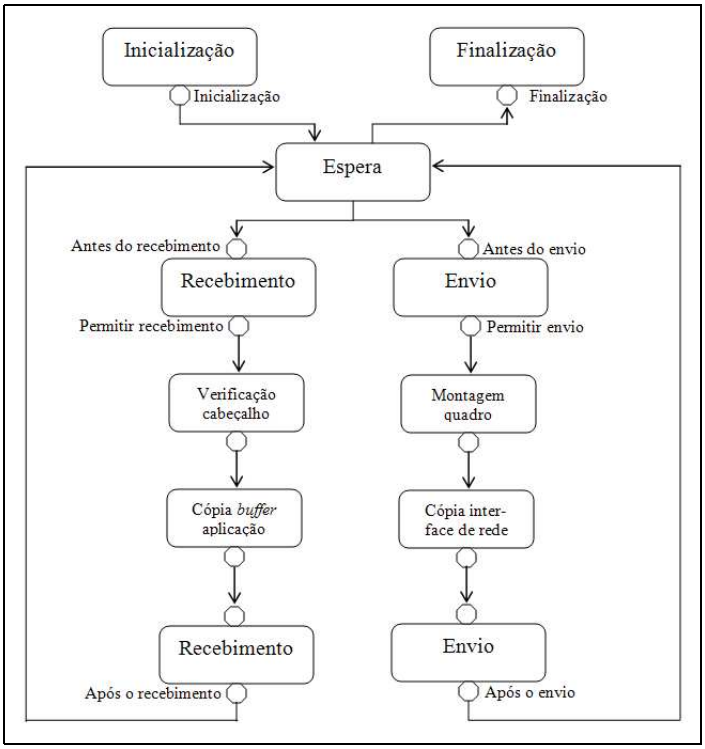

Figura 3.2: Fluxograma correspondente ao algoritmo de comunicação de um `Núcleo Básico` hipotético.

balhos de pesquisa que utilizam a metodologia de Programação Orientada a Aspectos, algoritmos complexos podem ser expressados através de abstrações ortogonais a outras abstrações do sistema. Assim como os aspectos na AOP agem sobre métodos espalhados pelas diversas classes de um sistema, as `Estratégias de Comunicação` vão agir nos `Pontos de Ação` definidos pelos `Núcleos Básicos`, geralmente localizados antes e depois de eventos importantes no algoritmo de comunicação.

`Estratégias de Comunicação` podem ser restritas a um determinado `Núcleo Básico` pois nem todos os `Núcleos Básicos` provêm os mesmos `Pontos de Ação`. Se uma determinada estratégia implementa algoritmos que agem em `Pontos de Ação` disponibilizados apenas por um determinado `Núcleo Básico`, essa estratégia não poderá combinar-se com outros `Núcleo Básicos` pois parte de sua implementação estará inacessível. A definição de `Pontos de Ação` genéricos e a utilização de uma interface comum para os `Núcleos Básicos` facilita a interoperabilidade entre `Estratégias de Comunicação` e `Núcleos Básicos`, possibilitando que uma mesma implementação para um determinado serviço de comunicação possa ser reutilizada em protocolos para diversas tecnologias de rede.

Cada `Estratégia de Comunicação` define três conjuntos de informações necessárias

para prover um determinado serviço: a implementação dos algoritmos relacionados ao serviço de comunicação, os atributos necessários para o armazenamento do estado referente a esses algoritmos e, além disso, todas as informações que a `Estratégia` deseja adicionar ao cabeçalho dos quadros trocados por instâncias do sistema de comunicação. Essas informações permitem que os algoritmos no nodo emissor e no nodo receptor troquem informações de controle entre si durante a comunicação.

3.1.3 `Framework`

O funcionamento do sistema de comunicação é dividido em duas etapas distintas: funcionamento em tempo de compilação e funcionamento em tempo de execução. Em tempo de compilação as `Estratégias de Comunicação` ativas são agrupadas por um componente que percorre o repositório de configuração e gera o `Protocolo Composto`, o protocolo de comunicação especializado a ser utilizado dinamicamente. O diagrama de ferrovias 3.3 ilustra o processo de geração de uma instancia do sistema de comunicação. Em tempo de execução o sistema de comunicação é constituído por apenas duas entidades: o `Núcleo Básico` correspondente à tecnologia de rede em uso e o `Protocolo Composto`. O `Núcleo Básico` faz chamadas a métodos do `Protocolo Composto` que por sua vez utiliza as implementações disponibilizadas pelas `Estratégias de Comunicação` ativas com o objetivo de prover os diferentes serviços de comunicação que essas `Estratégias` implementam. Diferentes combinações de `Estratégias` podem ser utilizadas para configurar o processo de comunicação.

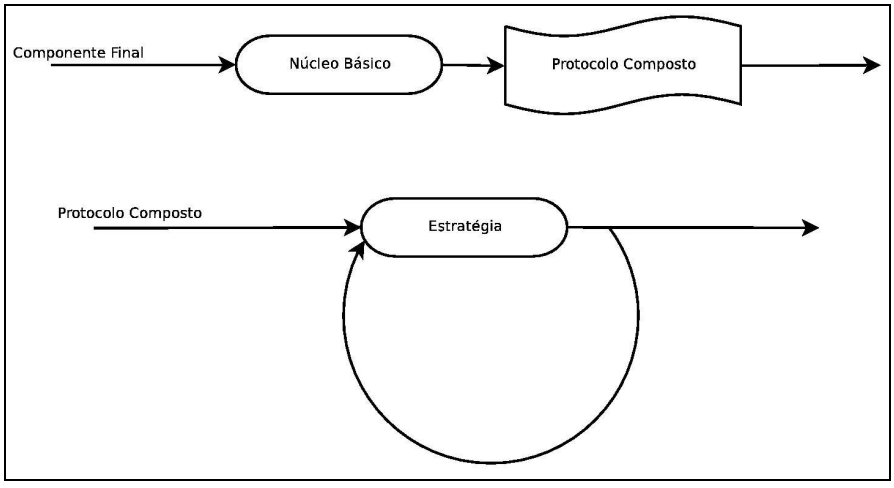

Figura 3.3: Geração de uma instancia do sistema de comunicação.

O `Protocolo Composto` tem acesso aos métodos e, conseqüentemente, ao estado do

Núcleo Básico. A relação entre o Núcleo Básico e o Protocolo Composto pode ser vista como uma dupla agregação, onde cada entidade possui uma referência à outra. Em linguagens que suportam essa construção, as duas classes podem ser declaradas como *friends* ou *package protected*, o que permite que o protocolo acesse os métodos protegidos do Núcleo Básico sem quebra de encapsulamento. A figura 3.4 apresenta o diagrama de classes referente ao sistema de comunicação em tempo de execução.

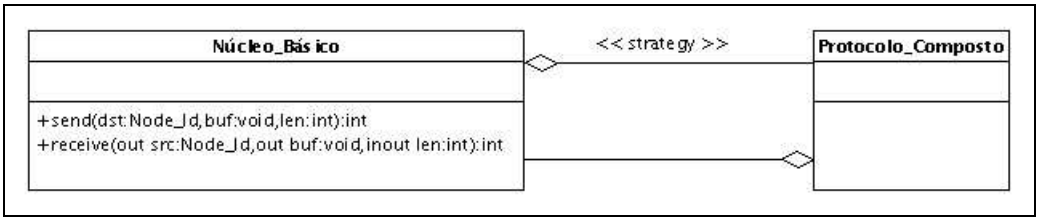

Figura 3.4: Diagrama de classes referente ao sistema de comunicação em tempo de execução.

O Núcleo Básico e o Protocolo Composto implementam uma variação do padrão de projeto *Strategy* [30], cuja estrutura está exemplificada na figura 3.5. O *Strategy* permite que diferentes membros de uma família de algoritmos que implementa uma determinada funcionalidade possam ser utilizados em um dado contexto, modificando o contexto de acordo com as suas implementações. O sistema de comunicação utiliza diferentes configurações de Protocolo Composto, variando de acordo com as Estratégias de Comunicação ativas, para modificar o comportamento do algoritmo de comunicação do Núcleo Básico.

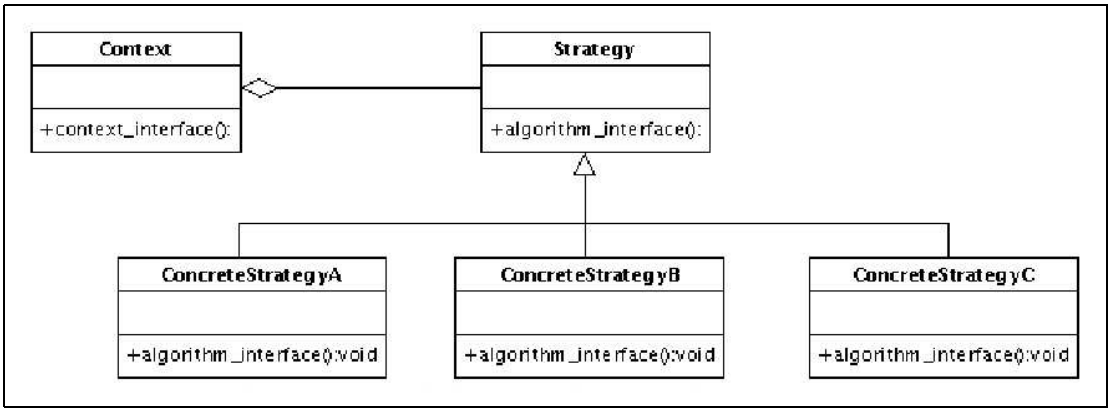

Figura 3.5: Diagrama de classes - *Strategy* [30].

A principal diferença entre a implementação tradicional do *Strategy* e a implementação utilizada no sistema de comunicação é o fato de as diferentes configurações para o Protocolo Composto serem geradas estaticamente por componentes meta-programados do Framework

do sistema de comunicação. O fato de o `Protocolo Composto` ser gerado estaticamente apresenta a desvantagem de impossibilitar que o protocolo seja reconfigurado dinamicamente. Entretanto, o `Framework` projetado para suportar a interação entre `Núcleos Básicos` e `Estratégias de Comunicação` pode ser facilmente estendido para suportar uma implementação tradicional do *Strategy* em conjunto com a implementação estática. Nesse cenário, o `Protocolo Composto` gerado em tempo de compilação seria apenas a configuração inicial do sistema e poderia ser substituído por outras configurações criadas dinamicamente.

Entretanto, por se tratar de um domínio onde os requisitos da aplicação são conhecidos antes da geração do sistema operacional e com o objetivo de otimizar ao máximo a implementação do sistema de comunicação, apenas a configuração estática foi utilizada. Implementações dinâmicas do *Strategy* utilizam polimorfismo que muitas vezes é menos eficiente do que uma implementação não-polimórfica. De fato, chamadas à métodos virtuais em C++, utilizadas na implementação de polimorfismo, podem ser até duas vezes mais dispendiosas do que chamadas a métodos não virtuais e até três vezes mais caras do que atribuições simples [31].

A combinação de `Estratégias` também é executada em tempo de compilação pelos componentes meta-programados do sistema de comunicação. O `Framework` do sistema de comunicação agrupa o conjunto de `Estratégias de Comunicação` ativas em um componente que foi projetado como uma variação do padrão de projeto *Composite*, cuja estrutura geral pode ser observada na figura 3.6. O padrão de projeto *Composite* é utilizado para lidar com um conjunto de objetos que implementam uma mesma interface como se fossem um só, agrupando o conjunto de objetos em uma lista e utilizando-os para prover a soma de suas funcionalidades. Quando um dos métodos do *Composite* é chamado, o *Composite* percorre os objetos adicionados a sua lista chamando os métodos equivalentes.

Usualmente novos elementos podem ser adicionados ou removidos do *Composite* em tempo de execução mas em diversas situações o *Composite* é o resultado do processo de geração de um componente e não vai ser alterado em tempo de execução. Com o objetivo de otimizar o desempenho do *Composite* utilizado para agrupar as `Estratégias`, o `Framework Meta-Programado` do sistema de comunicação tira proveito de características especiais da linguagem de programação C++ para gerar parte do `Protocolo Composto` em tempo de compilação, utilizando os algoritmos disponibilizados pelas `Estratégias de Comunicação` ativas. Durante o processo de geração do sistema os métodos do `Protocolo Composto` são preenchidos com as implementações das estratégias ativas, evitando o atraso relacionado às chamadas de métodos, geralmente virtuais, e ao percorrimento da lista utilizada para armazenar os obje-

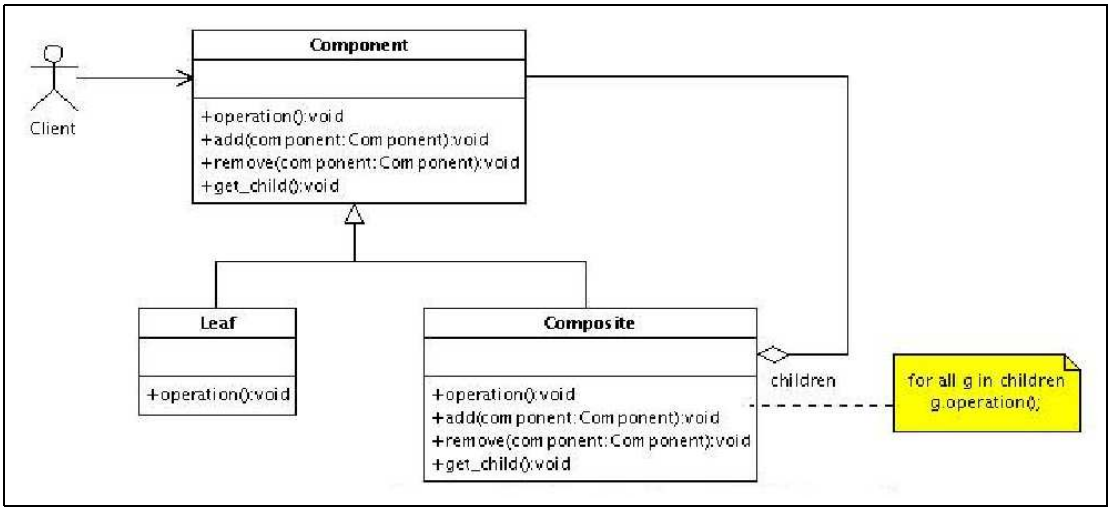

Figura 3.6: Diagrama de classes - *Composite* [30].

tos nas implementações dinâmicas do *Composite*. É importante observar que quando uma `Estratégias` é desativada ela não vai adicionar nenhum atraso ao processo de comunicação pois desativar uma `Estratégias` significa instruir o `Framework Meta-Programado` do sistema que o código para essa `Estratégias` não seja gerado.

Assim como no caso do *Strategy*, a implementação dinâmica do *Composite* apresentaria a vantagem de ser mais flexível. Entretanto, pelos mesmos motivos mencionado anteriormente, o sistema de comunicação disponibiliza apenas a implementação estática mas permite que a implementação dinâmica possa ser facilmente adicionada ao `Framework`.

3.2 `Framework Meta-Programado` do Sistema de Comunicação

Metaprogramação é uma tecnologia chave para o desenvolvimento de sistemas adaptáveis baseados em componentes. Metaprogramação estática com templates C++ é uma forma de metaprogramação limitada ao processo de compilação: o mecanismo de templates combinado com algumas outras características da linguagem C++ permite a criação de componentes que são resolvidos pelo compilador da linguagem. Esses componentes estaticamente meta-programados podem ser utilizados para manipular as características dos componentes dinâmicos do sistema e esse é o princípio básico de funcionamento do sistema de comunicação descrito nesse capítulo. O `Framework Meta-Programado` utilizado pelo sistema de comunicação utiliza componentes meta-programados em conjunto com outras funcionalidades da linguagem C++ com o objetivo de

prover mecanismos que permitem selecionar, configurar e combinar diferentes `Estratégias de Comunicação` de maneira eficiente.

Além disso, o repositório de configuração do sistema de comunicação também foi implementado através da utilização de um recurso disponibilizado pela linguagem C++: classes template *trait*. Classes *trait*, amplamente utilizadas para armazenar chaves de configuração na *Standard Template Library* (STL), podem ser vistas como pequenos componentes cujo propósito é agrupar informação utilizada por outros componentes [31]. O princípio de funcionamento de um repositório de configuração implementado através de classes *trait* é relativamente simples. Uma classe *trait* com informações de configuração gerais para o sistema é implementada como uma classe parametrizada. Para cada componente no sistema uma especialização dessa classe com chaves de configuração específicas para esse componente é projetada. Para acessar sua configuração, um determinado componente passa o seu tipo como parâmetro de classe para o repositório de configuração com o objetivo de identificar a classe *trait* responsável por armazenar as suas chaves de configuração.

Programação Gerativa é uma técnica que combina engenharia de domínio e metaprogramação com o objetivo de automatizar a criação de componentes de software [32]. Alguns dos componentes tradicionais de Programação Gerativa, mais especificamente a lista encadeada meta-programada (NODE) e as estruturas de repetição e geração de código (EFOR) e condição (IF), foram utilizados durante a implementação do `Framework Meta-Programado` do sistema de comunicação. Sunder e Musser [33] utilizam técnicas de meta-programação similares às utilizadas no desenvolvimento do componente descrito nessa seção para implementar suporte à programação orientada a aspectos. Além dos mecanismos fornecidos pela linguagem C++, o projeto também utilizou a biblioteca *Boost*, um framework meta-programado extensível que implementa funcionalidades equivalentes às da STL em tempo de compilação.

3.2.1 Estrutura das `Estratégias de Comunicação`

O principal objetivo do `Framework Meta-Programado` disponibilizado pelo sistema de comunicação é suportar o desenvolvimento de novas `Estratégias de Comunicação` e prover mecanismos que permitem combiná-las em um único componente, o `Protocolo Composto` a ser utilizado em conjunto com um `Núcleo Básico` em tempo de execução. Fazendo uma analogia entre a geração do `Protocolo Composto` e a construção de uma casa, as `Estratégias de Comunicação` poderiam ser vistas como os tijolos utilizados para cons-

truir as paredes da casa e o Framework Meta-Programado como o responsável por definir o lugar onde cada tijolo vai ser colocado e por fixar esses tijolos em seus devidos lugares. A figura 3.7 exemplifica a implementação de uma Estratégia de Comunicação.

```
template < bool active >
class Example_Strategy: public Strategies_Interface
{
protected:
    int attribute_1;  // (1) Atributos necessários para o funcionamento dos
                      // algoritmos referentes ao serviço de comunicação
public:
    struct Header
    {
        field_1;  // (2) Informações que esta Estratégia precisa adicionar
        field 1;  // ao cabeçalho dos quadros durante a comunicação
    };

    template < class T >
    static void init (T & context)
    {
         // (3) Código para a inicialização da Estratégia
    }

    template < class T >
    static void allow_send (T & context)
    {
         // (3) Código que implementa o serviço de comunicação
    }

    template < class T >
    static void allow_receive (T & context)
    {
         // (3) Código que implementa o serviço de comunicação
    }
};

template < >
class Example_Strategy < false >: public Strategy_Interface
{
public:
    struct Header { // Vazio };
};
```

Figura 3.7: Exemplo de Estratégia de Comunicação.

É importante observar que as classes utilizadas para definir as Estratégias de Comunicação não vão ser utilizadas da maneira convencional, ou seja, não vão ser instanciadas pelo sistema de comunicação, e por isso apresentam algumas características especiais. Além disso, os métodos definidos para uma determinada Estratégia de Comunicação são estáticos (palavra chave static na declaração do método) e *inline* (o corpo do método acompanha a sua declaração). Além disso, todos os métodos são declarados como template (template < class T >) e recebem como parâmetro uma referência ao contexto relativo à chamada desse método (T & context). Duas outras característica que chamam a atenção no exemplo são o fato de a classe que representa a Estratégia de Comunicação receber um parâmetro de classe que indica se a Estratégia está ou não ativa (template < bool active >) e o fato de uma especialização da classe template existir para o caso de o parâmetro active possuir

o valor false. Além disso, todas as Estratégias de Comunicação são especializações da classe Strategies_Interface. Todas essas convenções são necessária para que o processo de combinação das Estratégias de Comunicação ativas seja desempenhado pelo Framework Meta-Programado do sistema com a ajuda do compilador C++.

Devido ao fato de não serem utilizadas através de instanciação, as Estratégias de Comunicação não podem armazenar informações relativas ao estado de seus algoritmos. Entretanto, a implementação da maioria dos serviços de comunicação vai necessitar que informações referentes ao estado dos algoritmos associados à esse serviço sejam armazenadas em algum lugar. A seção 3.2.2 apresenta os mecanismos implementados pelo Framework Meta-Programado do sistema de comunicação responsáveis pela combinação das Estratégias de Comunicação, esclarece a motivação por trás das convenções adotadas para a definição dessas Estratégia e explica como o estado de cada Estratégia é armazenado em tempo de execução. A única responsabilidade das Estratégias de Comunicação em relação ao armazenamento de informações referentes ao estado de seus algoritmos é definir os atributos a serem utilizados para armazenar essas informações (1) e acessá-los através de uma referência recebida pelo protocolo composto.

No exemplo discutido anteriormente, dois campos vão ser adicionados ao cabeçalho de cada quadro durante a comunicação com o objetivo de controlar o funcionamento dos algoritmos implementados pela Estratégia de Comunicação (2). Além disso, o serviço de comunicação implementado pela Estratégia deve ser projetado de acordo com os Pontos de Ação que os Núcleos Básicos provêm (3).

3.2.1.1 Criação de Estratégias de Comunicação

De maneira a simplificar o desenvolvimento de novas Estratégias de Comunicação, o Framework Meta-Programado do sistema de comunicação provê uma classe que serve como interface para essas Estratégias. A classe Strategies_Interface define um conjunto de métodos vazios que correspondem aos Pontos de Ação definidos pelo conjunto de Núcleos Básicos suportados pelo sistema de comunicação. Ou seja, sempre que um determinado Núcleo Básico definir um novo Ponto de Ação, mesmo que esse Ponto de Ação seja projetado especificamente para esse Núcleo Básico, um novo método vazio deve ser adicionado a interface das Estratégias de Comunicação.

Diferente das implementações tradicionais de interfaces em C++, classes geralmente chama-

das de “puramente” abstratas, nas quais todos os métodos são puramente virtuais, os métodos da interface das Estratégias de Comunicação consistem em blocos vazios *inline* declarados como métodos estáticos. Essa é uma característica essencial para o funcionamento do Framework Meta-Programado do sistema de comunicação. Além de dificultar a implementação do Framework, declarar os métodos dessa interface como virtuais acarretaria em atrasos desnecessários no funcionamento dinâmico do sistema de comunicação já que todas as Estratégias de Comunicação vão ser resolvidas e combinadas estaticamente em tempo de compilação.

```
class Strategies_Interface
{
    public:
        template < class T >
        static void init (T & context) { // Vazio }

        template < class T >
        static void before_receive (T & context) { // Vazio }

        template < class T >
        static void allow_receive (T & context) { // Vazio }

        template < class T >
        static void receive_completed (T & context) { // Vazio }

        template < class T >
        static void after_receive (T & context) { // Vazio }

        template < class T >
        static void before_send (T & context) { // Vazio }

        template < class T >
        static void allow_send (T & context) { // Vazio }

        template < class T >
        static void send_completed (T & context) { // Vazio }

        template < class T >
        static void after_send (T & context) { // Vazio }

        template < class T >
        static void finalize (T & context) { // Vazio }
};
```

Figura 3.8: Interface das Estratégias de Comunicação.

Conforme discutido na seção anterior, todas as Estratégias de Comunicação declaram uma herança pública à interface das estratégias e redefinem os métodos correspondentes aos Pontos de Ação nos quais essas Estratégias vão agir. Todos os métodos não redefinidos por uma determinada Estratégia são herdados da interface, ou seja, todas as Estratégias de Comunicação possuem métodos correspondentes a todos os Pontos de Ação definidos pelo conjunto de Núcleos Básicos suportados pelos sistema de comunicação. Entretanto, a grande maioria desses métodos vai consistir em implementações vazias herdadas da interface das Estratégias. A figura 3.8 exemplifica uma implementação dessa interface, com métodos

correspondentes aos Pontos de Ação genéricos definidos na seção 3.1.1.

3.2.2 O Configurador de Protocolos e o Protocolo Composto

Como mencionado anteriormente, em tempo de execução o Núcleo Básico e o Protocolo Composto implementam uma variação do padrão de projeto *strategy*. A principal diferença entre o mecanismo utilizado nesse sistema de comunicação e a implementação tradicional do *strategy* é que o Framework Meta-Programado do sistema de comunicação gera o Protocolo Composto em tempo de compilação, combinando as Estratégias de Comunicação de acordo com as informações contidas no repositório de configuração.

A combinação das Estratégias é fruto da interação entre duas classes do Framework Meta-Programado: o Configurador de Protocolos e o Protocolo Composto. O Configurador de Protocolos é um componente meta-programado responsável por inicializar uma nova configuração de Protocolo Composto para um determinado Núcleo Básico de acordo com as informações lidas do repositório de configuração. O Protocolo Composto é uma classe meta-programada que é literalmente escrita pelo compilador C++ em tempo de compilação, de acordo com os parâmetros fornecidos pelo Configurador de Protocolos. Em tempo de compilação o Configurador de Protocolos percorre o repositório de configuração referente ao Núcleo Básico em questão e determina quais Estratégias de Comunicação foram ativadas. As classes referentes às Estratégias de Comunicação ativas são utilizadas para inicializar uma nova configuração de Protocolo Composto que vai ser utilizada em tempo de execução. Todo o processo de seleção, configuração e combinação estática de Estratégias de Comunicação está sumarizada na figura 3.9.

O Configurador de Protocolos estabelece o vínculo entre um determinado Núcleo Básico e as Estratégias de Comunicação que podem combinar-se com ele. Devido ao fato de que Estratégias de Comunicação podem ser projetadas especificamente para um determinado núcleo básico, um Configurador de Protocolos deve ser disponibilizado para cada Núcleo Básico presente no sistema de comunicação. Portanto, cada vez que um novo Núcleo Básico é adicionado ao sistema de comunicação um novo Configurador de Protocolos específico para esse Núcleo Básico deve ser criado. Como a implementação de configuradores de protocolos é relativamente simples (figura 3.10) e a estrutura geral da classe é a mesma para todos os Núcleo Básicos, variando apenas as Estratégias de Comunicação que são utilizadas, o fato de um novo configurador ter que ser criado para cada

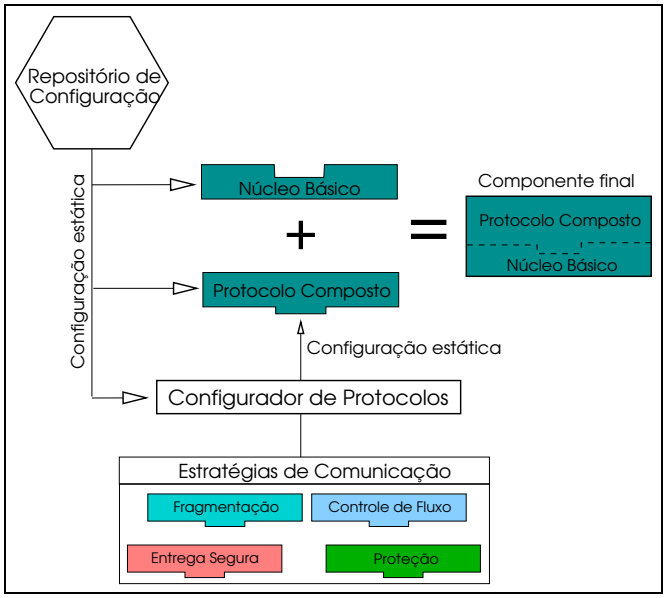

Figura 3.9: Seleção, configuração e combinação de `Estratégias` em tempo de compilação.

`Núcleo Básico` não chega a ser um problema. A complexidade relacionada à criação dessa classe é insignificante se comparada com a complexidade relacionada a definição e implementação dos algoritmos referentes às `Estratégias de Comunicação` e ao `Núcleo Básico`.

Um `Configurador de Protocolos` recebe como parâmetro de classe o tipo referente ao `Núcleo Básico` correspondente a esse configurador (1). Essa informação é utilizada para definir o repositório de configuração a ser utilizado (3) e também para inicializar o `Protocolo Composto` (4), que precisa saber qual `Núcleo Básico` vai ser utilizado em tempo de execução.

```
template < class Baseline >   // (1)
struct Protocol_Configurator
{
    typedef Composite_Protocol <
        // (2)
        Node < Multicast_Strategy <Traits < Baseline >::MULTICAST>,  // (3)
        Node < Fragment_Strategy < Traits < Baseline >::FRAGMENT >,
        Node < Flow_Control_Strategy < Traits < Baseline >::FLOW_CONTROL >,
        Node < Reliable_Delivery_Strategy < Traits < Baseline >::RELIABLE_DELIVERY >
        > > > >, Baseline  // (4)
    > Protocol;
};
```

Figura 3.10: Exemplo de `Configurador de Protocolos`.

Em tempo de compilação, o `Configurador de Protocolos` cria uma lista encadeada meta-programada onde os elementos de cada nodo são as classes referentes às `Estratégias de Comunicação` que podem combinar-se com o `Núcleo Básico` em uso (2). A ordem em que essas classes são adicionadas a lista encadeada é um fator importante pois define a ordem de ativação das `Estratégias de Comunicação` em tempo de execução. Existem situações

onde a ativação de um conjunto de Estratégias de Comunicação em uma ordem arbitrária pode prejudicar o desempenho ou até mesmo inviabilizar completamente o processo de comunicação e por esse motivo a ordem de ativação das Estratégias deve ser definida com cuidado, levando-se em consideração os algoritmos das Estratégias de Comunicação a serem combinadas.

É importante observar que a classe que o Configurador de Protocolos seleciona para cada Estratégia de Comunicação depende da chave boleana correspondente a essa Estratégia no repositório de configuração (3). Se a chave for definida com o valor verdadeiro, indicando que essa determinada Estratégia está ativa, a implementação real da Estratégia é adicionada à lista encadeada. Se a chave estiver definida com o valor falso, a implementação vazia que essa Estratégia de Comunicação define é adicionada à lista encadeada. Se a chave ou qualquer uma das classes utilizadas pelo Configurador de Protocolos não existir, um erro de compilação é lançado.

Fica claro com a explicação do funcionamento do Configurador de Protocolos porque todas as classes referentes a Estratégias de Comunicação devem definir uma especialização para o caso de seu parâmetro de classe possuir o valor falso, conforme discutido na seção 3.2.1. Essa especialização consiste em uma classe vazia, onde todos os métodos são blocos *inline* vazios herdados da interface das Estratégias, nenhum atributo é definido e nenhum campo é adicionado ao cabeçalho dos quadros. Essas classes vão ser adicionadas a lista encadeada no lugar das Estratégias que não foram ativadas, de maneira que nenhum código referente a uma Estratégia que não tenha sido ativada esteja presente no Protocolo Composto em tempo de execução.

A classe Protocolo Composto aceita como parâmetros de classe uma lista contendo as Estratégias de Comunicação ativas e o tipo referente ao Núcleo Básico em uso (figura 3.11), ambos os parâmetros são fornecidos pelo Configurador de Protocolos. A lista de Estratégias de Comunicação vai ser utilizada pelo Protocolo Composto para acessar os métodos, atributos e campos de cabeçalho definidos por essas Estratégias. O tipo referente ao Núcleo Básico é necessário pois existe apenas uma classe Protocolo Composto para todas as combinações de Núcleo Básico e Estratégias de Comunicação possíveis e essa classe precisa saber qual Núcleo Básico está sendo utilizado.

Conforme pode ser observado na figura, o Protocolo Composto declara uma herança pública ao tipo RESULT definido pela classe Merge_Strategies, uma classe parametrizada

```
template < class List, class Baseline_Architecture >
class Composite_Protocol: public Merge_Strategies < List >::RESULT // (1)
```

Figura 3.11: Declaração da classe `Protocolo Composto`.

que recebe a lista encadeada de `Estratégias de Comunicação` como parâmetro (1). Essa classe cria uma cadeia de herança entre todas as `Estratégias de Comunicação` presentes na lista encadeada e associa ao tipo `RESULT` essa cadeia de herança. Ou seja, o `Protocolo Composto` é uma sub-classe de todas as `Estratégias de Comunicação` definidas pelo `Configurador de Protocolos` e conseqüentemente herda todos os atributos definidos por essas `Estratégias`. Isso é necessário para que o estado do algoritmo das `Estratégias de Comunicação` possa ser salvo em tempo de execução: as `Estratégias` em si não têm estado, mas definem atributos que são herdados pelo `Protocolo Composto` que existe em tempo de execução e portanto pode armazenar informações referentes ao estado do algoritmos implementados por essas `Estratégias`. A figura 3.12 apresenta a implementação da classe `Merge_Strategies`. Note que essa classe utiliza os mecanismos de recursão e especialização de templates para implementar uma cadeia de herança entre as classes na lista passada como parâmetro.

```
struct Empty_Class
{
};

struct _Empty_Class
{
};

template < int from, int to, class Class_List >
struct _Merge_Strategies
{
    struct RESULT: public IF < Less::Compare < from, to >::RESULT,
                       typename Get < Class_List, from >::RESULT,
                       Empty_Class >::RESULT,
                   public IF < Less::Compare < from, to >::RESULT,
                       typename _Merge_Strategies < from + 1, to, Class_List >::RESULT,
                       _Empty_Class >::RESULT { };
};

template < class Class_List >
struct Merge_Strategies
{
    struct RESULT: public IF < Less::Compare < 0, Length < Class_List >::RESULT >::RESULT,
                       typename Get < Class_List, 0 >::RESULT,
                       Empty_Class >::RESULT,
                   public IF < Less::Compare < 0, Length < Class_List >::RESULT >::RESULT,
                       typename _Merge_Strategies <
                           1, Length < Class_List >::RESULT, Class_List >::RESULT,
                       _Empty_Class >::RESULT { };
};
```

Figura 3.12: Implementação da classe `Merge_Strategies`.

Um componente similar a `Merge_Strategies` é utilizado para criar uma cadeia de herança

entre os campos de cabeçalho definidos pelas diferentes estratégias na lista encadeada criada pelo Configurador de Protocolos. Esse cabeçalho deve ser adicionado a cada um dos quadros enviados pelo Núcleo Básico para que as estratégias de comunicação ativas possam trocar informações entre o nodo emissor e o nodo receptor durante a comunicação.

O Protocolo Composto define os métodos a serem chamados pelos Núcleos Básicos em seus Pontos de Ação. Assim como na interface das Estratégias de Comunicação, métodos correspondentes a todos os Pontos de Ação definidos por todos os Núcleos Básicos estão presentes no Protocolo Composto. Para cada método definido pelo Protocolo Composto, a meta-função Call_Protocols, um mecanismo estático de geração de código, é utilizada para percorrer a lista de Estratégias de Comunicação ativas gerando código para a chamada de método correspondente de cada Estratégia de Comunicação. Como todos os métodos definidos pelas Estratégias de Comunicação são declarados como *inline*, o compilador C++ substitui a chamada de método pelo corpo do método na etapa de otimização, evitando o atraso associado a chamadas de métodos. A meta-função Call_Protocols utiliza a estrutura de repetição e geração de código EFOR e recebe como parâmetro um Statement, uma classe utilizada para indicar qual o método a sendo chamado. A figura 3.13 apresenta a implementação dessa classe.

```
template < class Statement >
struct Call_Protocols
{
    static void exec (Environment & context)
    {
        EFOR < 0, Less, Length < List >::RESULT, +1, Statement >::exec (context);
    }
};
```

Figura 3.13: Implementação da meta-função Call_Protocols.

Classes Statement correspondentes a cada um dos Ponto de Ação definido pelo conjunto de Núcleos Básicos do sistema de comunicação devem ser criadas de maneira a permitir que o Protocolo Composto possa utilizar a meta-função Call_Protocols na implementação de seus métodos. Ou seja, para cada Ponto de Ação adicionado por um determinado Núcleo Básico novos métodos tem que ser criados para interface das Estratégias, discutida na seção anterior, e para o Protocolo Composto e um novo Statement deve ser implementado. A figura 3.14 apresenta a implementação do Statement para o Ponto de Ação de inicialização.

De maneira a permitir que as Estratégias de Comunicação acessem o estado dinâmico do Núcleo Básico e dos algoritmos definidos por elas, cada método definido pe-

```
struct Init
{
    template < int i > struct Body
    {
        static void exec (Protocols_Environment & context)
        {
            Get < Protocols_List, i >::RESULT::init (context);
        }
    };
};
```

Figura 3.14: Implementação do `Statement` para o `Ponto de Ação` de inicialização.

las `Estratégias` recebe como parâmetro um objeto que armazena informações referentes ao contexto da chamada de método. Esse objeto é uma instância da classe `Protocols_Context`, que possui uma referência ao `Protocolo Composto`, utilizada por cada `Estratégia` para acessar o estado dinâmico de seus algoritmos, e uma referência ao `Núcleo Básico`. Além disso, essa classe possui outros atributos que são utilizados para estabelecer um mecanismo de comunicação entre o `Núcleo Básico`, `Protocolo Composto` e `Estratégias de Comunicação`, além da comunicação entre as `Estratégias de Comunicação` ativas. A figura 3.15 apresenta a implementação da classe.

```
template < class Composite_Protocol, class Baseline_Architecture >
class Protocols_Context
{
public:
    Composite_Protocol * protocol;
    Baseline_Architecture * baseline;
    int allow_send, allow_receive;
    int send_completed, receive_completed;
    unsigned int dst;
    void * sbuf;
    unsigned int slen;
    unsigned int * src;
    void * rbuf;
    unsigned int * rlen;

    void init (Composite_Protocol * protocol, Baseline_Architecture * baseline)
    {
        this->protocol = protocol;
        this->baseline = baseline;
    }
};
```

Figura 3.15: Implementação da classe `Protocols_Context`.

É importante observar que as `Estratégias` devem acessar os seus próprios atributos através da referência ao `Protocolo Composto` obtida através do contexto passado como parâmetro a cada um de seus métodos. Esse é um recurso que pode parecer estranho a primeira vista mas é indispensável já que as `Estratégias de Comunicação` não existem em tempo de execução.

A figura 3.16 apresenta os métodos definidos para o `Protocolo Composto` que correspondem a alguns dos `Pontos de Ação` padrão. Note que em alguns métodos informações são

salvas ou lidas do contexto passado por parâmetro para os métodos das `Estratégias`. Note também que a função meta-programada `Call_Protocols` é utilizada para gerar o código para a chamada dos métodos das `Estratégias`. Como esses métodos são *inline*, o corpo dos métodos vai substituir a chamada e no caso dos métodos vazios é como se esses não existissem.

```
void init (Baseline_Architecture & user)
{
    context.init (this, user);
    Call_Protocols < typename Statements::Init >::exec(context);
}

void before_receive (unsigned int * src, void * buf, unsigned int * len)
{
    context.src = src;
    context.rbuf = buf;
    context.rlen = len;
    Call_Protocols < typename Statements::Get_Before_Receive >::exec(context);
}

int allow_receive ()
{
    Call_Protocols < typename Statements::Get_Allow_Receive >::exec(context);
    return context.allow_receive;
}

int receive_completed ()
{
    Call_Protocols < typename Statements::Get_Receive_Completed >::exec(context);
    return context.receive_completed;
}

void after_receive ()
{
    Call_Protocols < typename Statements::Get_After_Receive >::exec(context);
}
```

Figura 3.16: Métodos do `Protocolo Composto`.

Capítulo 4

Implementação Myrinet/EPOS

Com o objetivo de validar os conceitos apresentados no capítulo 3, o sistema de comunicação descrito foi implementado no ambiente de Programação Paralela SNOW. O `Framework Meta-Programado` do sistema de comunicação foi integrado ao EPOS, o sistema operacional especializado utilizado no projeto SNOW, e um `Núcleo Básico` foi desenvolvido para a tecnologia de rede Myrinet, a *System Area Network* que interconecta os nodos do SNOW. Além disso, aplicações que implementam o *benchmark* Ping-Pong foram desenvolvidas para o EPOS e `Estratégias de Comunicação` que implementam protocolos leves para controle de fluxo, entrega segura e fragmentação foram projetadas e implementadas.

As próximas seções discutem os detalhes referentes a essas implementações, concentrando-se na arquitetura do `Núcleo Básico` Myrinet e no desempenho dos protocolos leves desenvolvidos. Os detalhes referentes à integração do sistema de comunicação com o sistema operacional EPOS também são abordados.

4.1 A Tecnologia de Rede Myrinet

Com o objetivo de compreender o projeto e a implementação do `Núcleo Básico` desenvolvido para a tecnologia de rede Myrinet é preciso entender as características dessa tecnologia de rede. A Myrinet, fabricada pela empresa Myricom e padronizada pelo instituto ANSI sob a norma *Myrinet-on-VME Protocol Specification* [34], consiste em uma *System Area Network* (SAN) onde cada nodo possui uma interface de rede Myrinet conectada ao seu barramento de entrada e saída. A interface de rede é utilizada, em conjunto com *links* e *switches* de alto desempenho e com baixíssimas taxas de erro, para interconectar os nodos da SAN entre si. Em 2005, cerca de 28%

dos 500 sistemas computacionais de alto desempenho mais eficientes do mundo utilizam redes Myrinet para interconectar os seus processadores (figura 4.1).

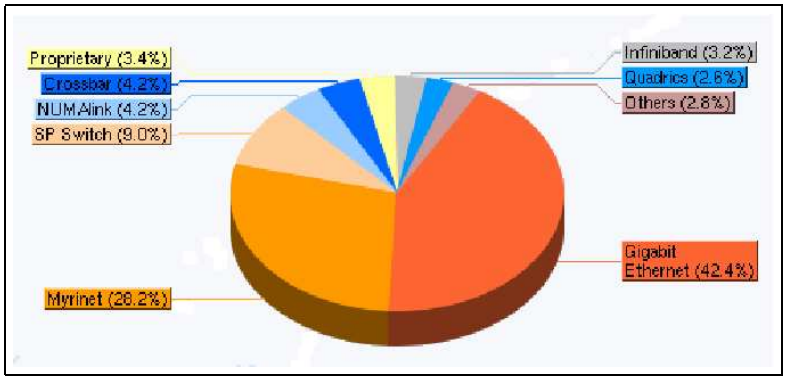

Figura 4.1: Tecnologias de interconexão utilizadas pelos 500 supercomputadores mais eficientes do mundo [2].

4.1.1 Estrutura dos Quadros e Mecanismos de Roteamento

O padrão Myrinet define quadros de tamanho variável que podem encapsular outros pacotes, incluindo pacotes IP e quadros Ethernet, sem a necessidade de utilizar camadas de adaptação. Cada quadro é identificado por um tipo específico, o que permite que diversos protocolos de enlace Myrinet possam transmitir quadros concorrentemente em uma mesma SAN. A figura 4.2 apresenta a estrutura de um quadro Myrinet, como definida no padrão ANSI.

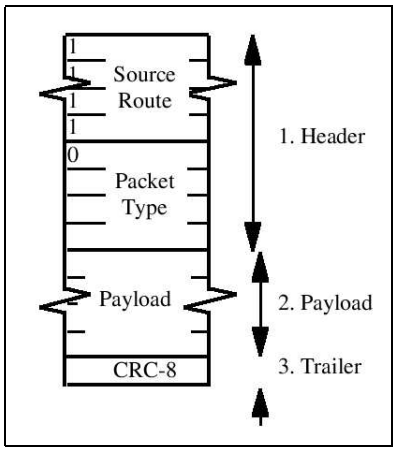

Figura 4.2: Estrutura de um quadro Myrinet [34].

O cabeçalho de um quadro Myrinet é constituído pelo campo *Source Route*, onde os bytes de roteamento do quadro são definidos, e pelo campo *Packet Type*, que identifica o tipo associado ao quadro. O tamanho do cabeçalho varia de acordo com a quantidade de bytes de roteamento, que

depende do número de *switches* Myrinet no caminho entre o nodo transmissor e o nodo receptor. Durante o roteamento, cada *switch* por onde um quadro passa remove o primeiro byte de roteamento do quadro e o encaminha para a porta indicada nesse byte, se essa porta estiver livre. Caso a porta se encontre bloqueada ou ocupada o quadro é armazenado até a liberação da porta ou até o final de um período de espera que pode ser configurado (veja controle de fluxo *Backpressure* na seção 4.1.3).

Se a porta destino especificada pelos bytes de roteamento de um quadro estiver livre, o *switch* Myrinet vai encaminhar esse quadro assim que o byte de roteamento é decodificado, mesmo que o quadro não tenha sido recebido inteiramente. Esse mecanismo é chamado de *Wormhole Routing* e é utilizado para aumentar o desempenho do roteamento. O bit mais significativo de cada um dos bytes de roteamento deve obrigatoriamente possuir o valor 1, o que permite que os *switches* identifiquem problemas de roteamento ou de criação de cabeçalho e descartem quadros errôneos.

O campo *Packet Type* é utilizado para identificar o protocolo de enlace responsável por tratar o quadro. Esse campo é constituído por 4 bytes e o bit mais significativo do primeiro desses bytes deve obrigatoriamente possuir o valor 0. Apesar de o cabeçalho de um quadro Myrinet não ter tamanho definido é sempre possível identificar o campo *Packet Type* pelo valor do bit mais significativo do seu primeiro byte, cujo valor difere dos bits mais significativos dos bytes de roteamento.

O formato e interpretação do campo *Payload* de um quadro Myrinet dependem dos protocolos definidos para os dados sendo transportados. Geralmente o campo *Payload* é utilizado para encapsular mensagens de outros protocolos que definem os seus próprios cabeçalhos, como por exemplo quadros Ethernet ou pacotes IP, e por esse motivo o cabeçalho de um quadro Myrinet possui o formato simplificado descrito anteriormente. O campo *Trailer* contém o *Cyclic-Redundancy-Check* de 8 bits (CRC-8) referente aos outros bytes que constituem o quadro. O CRC-8 identifica quadros cujos dados foram corrompidos por falhas no hardware de comunicação.

É importante observar que as técnicas de roteamento utilizadas pela tecnologia de rede Myrinet, apesar de eficientes, apresentam uma desvantagem em relação à técnicas mais simples como a utilizada por *switches* Ethernet: o hardware de comunicação não suporta mensagens *broadcast*. De fato, as implementações eficientes de *broadcast* e *multicast* para a tecnologia de rede Myrinet são bastante complexas e têm sido um tema de pesquisa bastante ativo [35, 36].

4.1.2 Topologia

A topologia de uma rede Myrinet pode ser vista como um grafo conectado e não-direcionado, onde as interfaces de rede e os *switches* constituem os vértices e as ligações entre interfaces e *switches* constituem as arestas. Qualquer combinação de vértices e arestas é válida desde que o grafo permaneça conectado. O grafo pode conter ciclos, o que é recomendado para redes de grande porte pois os diversos caminhos redundantes para os nodos da rede representados pelos ciclos permitem que esses nodos sejam acessíveis mesmo em caso de falhas no hardware de comunicação. A figura 4.3 apresenta exemplos de possíveis topologias para uma rede Myrinet. A topologia A é a menor topologia possível, contendo apenas dois nodos ligados entre si através de suas interfaces de rede.

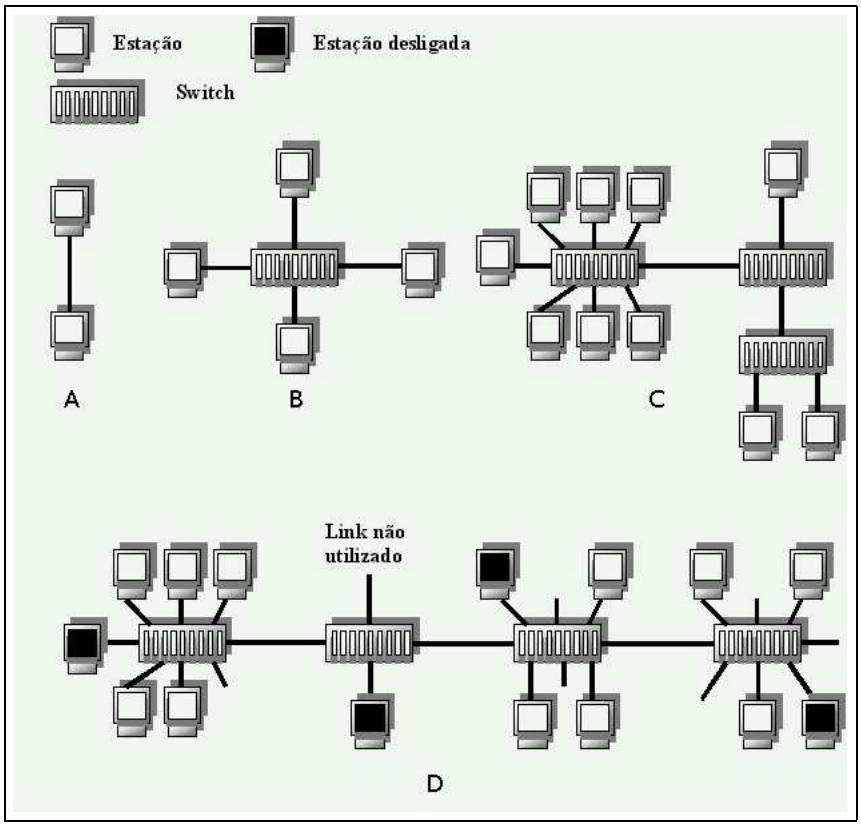

Figura 4.3: Possíveis topologias em uma SAN Myrinet.

4.1.3 Placa de Interface de Rede (NIC)

A NIC Myrinet é um dispositivo de comunicação programável, implementado em torno de um processador de rede, que serve como interface a uma SAN Myrinet. Cada interface de rede Myrinet possui de 2 a 4 MB de memória SRAM, dependendo do modelo, e um conjunto de três contro-

ladores de DMA responsáveis pela transmissão de dados entre host, NIC e SAN. Além disso, as interfaces Myrinet disponibilizam um processador RISC embarcado responsável por executar um programa de controle que é chamado de *Myrinet Control Program* (MCP). O MCP pode ser utilizado pelos desenvolvedores de protocolos para implementar serviços de comunicação variados que são executados pelo processador da interface de rede, aliviando o host da carga de processamento relacionado aos serviços de comunicação. É importante observar que o processador da interface de rede Myrinet é muito simples e normalmente apresenta um desempenho inferior ao processador utilizado pelo host. Por esse motivo, o MCP empregado em um determinado sistema de comunicação deve ser projetado com cautela pois adicionar apenas algumas instruções no caminho crítico do algoritmo de comunicação pode impactar o desempenho do sistema.

A tecnologia de rede Myrinet exige que todos os quadros enviados ou recebidos por um nodo passem pela memória SRAM da interface de rede, pois os controladores de DMA responsáveis pelo envio e recebimento de quadros só tem acesso à memória da NIC. Além disso, a memória SRAM da interface de rede é utilizada para armazenar o MCP e todas as estruturas de dados utilizadas durante a comunicação, como por exemplo as estruturas utilizadas pelo controlador de DMA responsável pela transferência de dados entre host e NIC.

Em uma rede Myrinet existem duas maneiras de transferir dados da memória principal do host para a memória da interface de rede e vice-versa: Programmed I/O (PIO) e o controlador de DMA `Host/NIC` da interface de rede Myrinet. Fatores como a utilização de *write-combining* [37], o atraso associado a inicialização de um DMA e o fato de que para cada DMA `Host/NIC` pelo menos 24 bytes referentes a estrutura de controle de operações de DMA têm que ser escritos na memória SRAM da interface de rede faz com que PIO seja mais eficiente do que DMA para transferência de quadros pequenos entre a memória principal do host e a memória da NIC. Além disso, devido ao fato de operar com endereços físicos, fatores como a tradução de endereços lógicos para endereços físicos e o *swapping* da memória para o disco rígido têm que ser observados durante as operações de DMA.

A figura 4.4 apresenta o atraso associado aos dois mecanismos de transmissão de dados entre host e interface de rede em um nodo de rede Myrinet. A transferência de dados da memória da NIC para a memória principal do host é sempre mais eficiente se for feita através do controlador de DMA já que leituras efetuadas utilizando PIO apresentam um desempenho inferior em relação a escritas que utilizam PIO [38, 37].

Os três controladores de DMA disponibilizados pela interface de rede Myrinet são chamados de `Net-Send`, responsável por injetar quadros na rede, `Net-Receive`, responsável por consu-

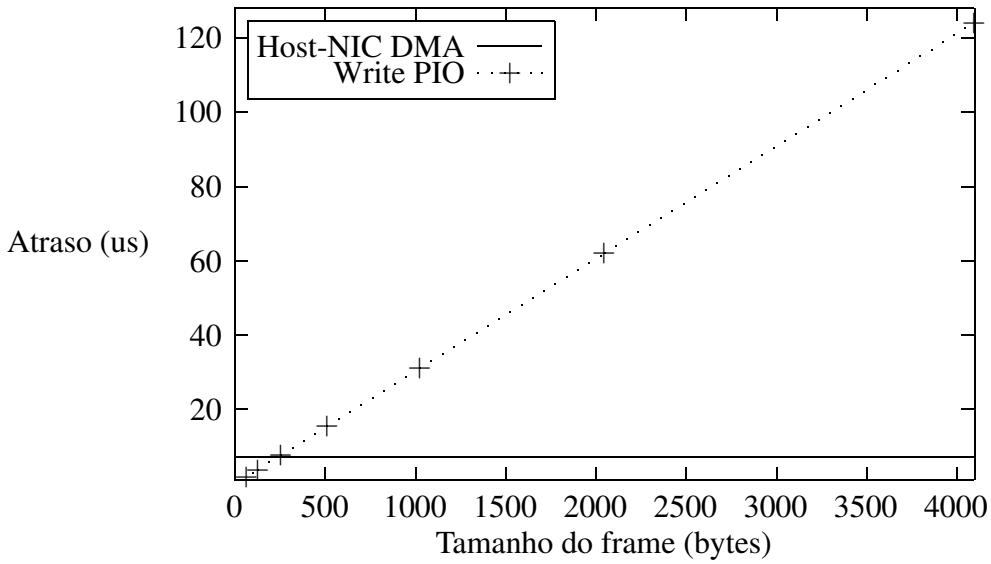

Figura 4.4: Atraso associado aos dois mecanismos de transmissão de dados entre host e interface de rede Myrinet.

mir quadros recebidos da rede e `Host/NIC`, responsável pela transferência de dados entre host e interface de rede. Esses controladores de DMA podem ser programados para trabalhar em paralelo com o objetivo de aumentar o desempenho da comunicação. O número de operações de DMA que podem ser executadas em paralelo é limitado apenas pela restrição no número de acessos a memória SRAM da interface de rede em cada ciclo de relógio. A NIC Myrinet impõe uma ordem de prioridade para esses acessos: host, através de *Programmed I/O* ou através do controlador de DMA `Host/NIC`, o controlador de DMA `Net-Receive`, o controlador de DMA `Net-Send` e o processador da interface de rede. A figura 4.5 apresenta o diagrama de blocos de uma NIC Myrinet. Os controladores de DMA responsáveis pela interface com a SAN Myrinet estão localizados no bloco *Packet Interface* e o controlador responsável pela transferência de dados entre a memória do host e a memória da placa está localizado no bloco *PCIDMA chip*.

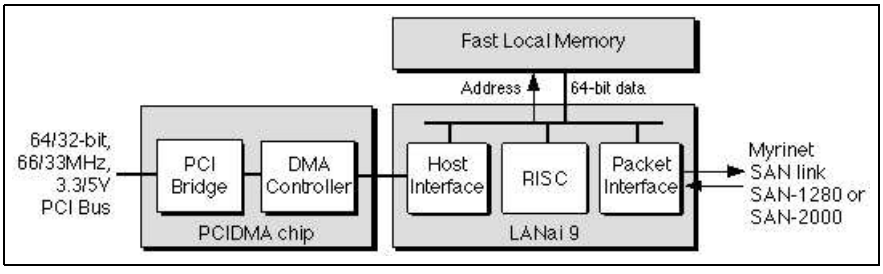

Figura 4.5: Diagrama de blocos de uma interface de rede Myrinet [34].

Assim como a grande maioria das interfaces de rede Ethernet, as NICs Myrinet conectam-se ao

barramento PCI ou SBUS do host e podem ser configuradas para trabalhar tanto em barramentos de 32 bits quanto em barramentos de 64 bits, com freqüências que variam de 33 MHz a 200 MHz [39]. A figura 4.6 exibe a arquitetura de um nodo em uma SAN Myrinet.

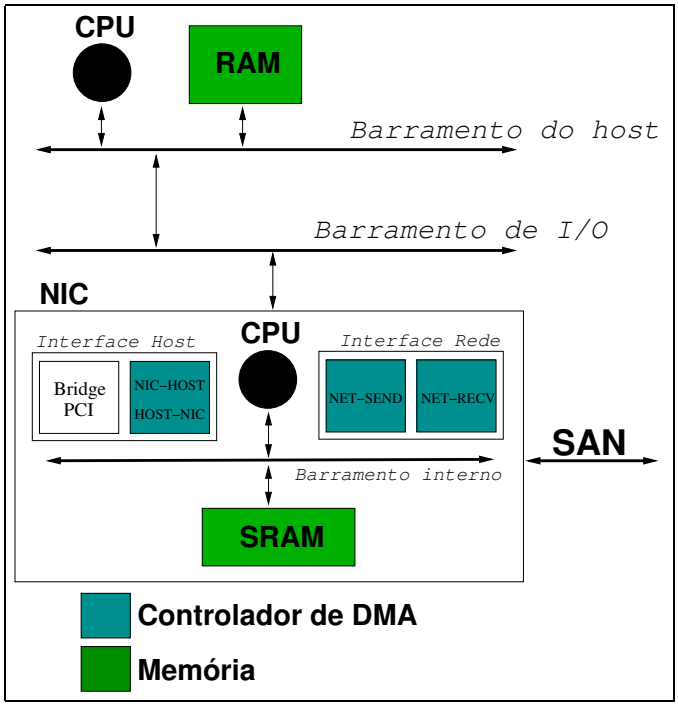

Figura 4.6: Arquitetura de um nodo em uma SAN Myrinet.

A tecnologia de rede Myrinet apresenta diversas outras funcionalidades que podem ser exploradas para aumentar o desempenho da comunicação:

Cadeias de DMA Host/NIC: O controlador de DMA `Host/NIC` utiliza estruturas de controle lidas da memória SRAM da interface de rede para coordenar as operações de transferência de dados entre a memória principal do Host e a memória da NIC. Novas requisições de transferência de dados podem ser encadeadas na memória da interface de rede, permitindo que as operações de DMA entre Host e NIC sejam efetuadas assincronamente pelo controlador de DMA.

Mecanismo de *Doorbell* implementado em hardware: A NIC Myrinet provê um mecanismo que permite que os dados escritos pelo host em qualquer área de uma região de I/O específica, chamada de região de *Doorbell*, sejam armazenados em uma fila FIFO circular na memória da interface de rede. O endereço de I/O utilizado pelo host para acionar o mecanismo de *Doorbell* também é armazenado nessa fila circular.

Controle de fluxo *Backpressure*: Controle de fluxo ponto-a-ponto empregado pela rede Myrinet que faz com que o nodo emissor espere por um período pré-determinado de tempo até que o nodo receptor esteja disponível para receber a mensagem. O tempo máximo de espera, que pode ser configurado dinamicamente por software, é utilizado para prevenir *deadlocks* na rede. Se o nodo receptor não estiver apto a receber o quadro e o tempo de espera se encerrar, o quadro é truncado ou a interface de rede do nodo receptor é re-inicializada.

4.2 `Núcleo Básico` Myrinet

Simplicidade e alto desempenho foram os dois principais requisitos que nortearam o projeto do `Núcleo Básico` para a tecnologia de rede Myrinet. Alto desempenho é uma característica imprescindível pois as diversas `Estratégias de Comunicação` que se combinam com o `Núcleo Básico` aumentam a complexidade, e conseqüentemente afetam o desempenho, do algoritmo de comunicação. Se o `Núcleo Básico` não disponibilizasse um desempenho aceitável a utilização de `Estratégias` com o objetivo de prover os serviços de comunicação requisitados pelas aplicações seria inviável. A simplicidade do `Núcleo Básico` Myrinet, além de ser um dos fatores que contribuem com o desempenho do sistema, facilita o projeto e implementação de `Estratégias de Comunicação`. De fato, a implementação da maioria dos serviços de comunicação foi deixado de fora do `Núcleo Básico` Myrinet, o que permite que uma ampla gama de `Estratégias de Comunicação` seja desenvolvida. Como as `Estratégias` de comunicação podem ser adicionadas ou removidas do sistema de comunicação de acordo com os requisitos das aplicações, o fato de o `Núcleo Básico` delegar a implementação de diversos serviços de comunicação às `Estratégias` contribui com a configurabilidade do sistema.

4.2.1 Arquitetura

Um dos fatores responsáveis pela degradação do desempenho de sistemas de comunicação tradicionais é o número excessivo de cópias durante o envio e o recebimento de mensagens. O `Núcleo Básico` desenvolvido para a tecnologia de rede Myrinet foi projetado de maneira a utilizar um número reduzido de cópias durante o envio e recebimento de quadros. Além disso, um mecanismo que leva em consideração o tamanho do quadro é empregado para efetuar a transferência de dados entre a memória principal do host e a memória SRAM da NIC. Em transferências do host para a NIC, PIO é utilizado para quadros pequenos e DMA é utilizado para quadros gran-

des. O tamanho que caracteriza um quadro pequeno é configurável e pode ser re-definido dinamicamente. Devido ao fato de operações PIO só poderem se efetuadas pelo host, e ao desempenho não satisfatório de leituras efetuadas através de PIO, o controlador de DMA é utilizado em todas as transferências de dados da NIC para o host. A figura 4.7 apresenta um diagrama referente ao `Núcleo Básico` projetado para a tecnologia de rede Myrinet, destacando as estruturas de dados utilizadas pelo nodo transmissor e o nodo receptor, assim como as cópias de quadros durante o processo de comunicação [4].

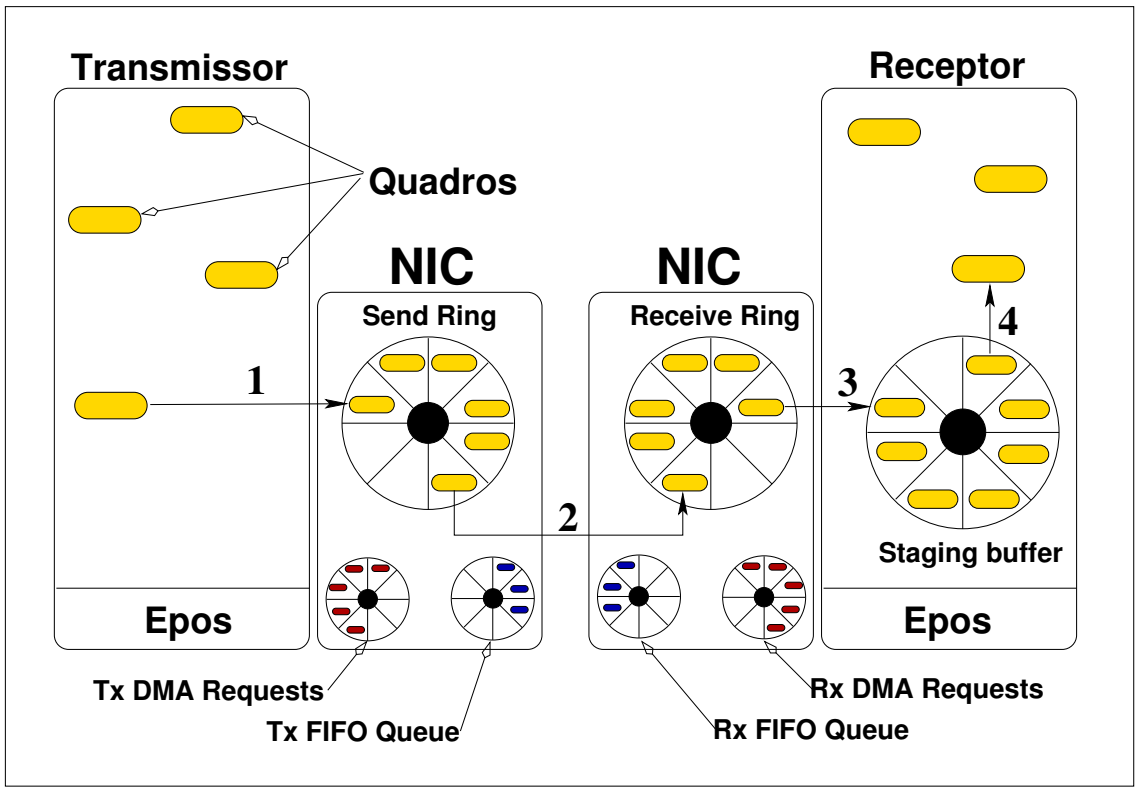

Figura 4.7: Estruturas de dados e cópias de quadros no `Núcleo Básico` Myrinet.

A memória SRAM disponibilizada pela interface de rede Myrinet é utilizada para armazenar a maioria dos *buffers* utilizados durante o envio e recebimento de mensagens. `Send Ring` e `Receive Ring` são *buffers* circulares onde os quadros são armazenados para que possam ser acessados pelo controlador de DMA responsável pela transmissão da NIC para a rede e vice-versa. `Rx DMA Requests` e `Tx DMA Requests` são cadeias circulares de estruturas de dados responsáveis pelo controle dos processos de DMA entre host e NIC no envio e no recebimento de mensagens. `Rx FIFO Queue` e `Tx FIFO Queue` são filas *First-In-First-Out* circulares utilizadas pelo processador do host e pelo processador da NIC Myrinet para sinalizar um para o outro a chegada de um novo quadro. A escolha do tamanho desses *buffers*, fator que pode afetar o desem-

penho da comunicação, depende das características da aplicação, do sistema operacional em uso e das Estratégias de Comunicação ativas. O Núcleo Básico Myrinet foi projetado de maneira a permitir que o tamanho de todas as estruturas utilizadas durante a comunicação seja dinamicamente configurável.

Para cada quadro a ser transmitido durante o processo de comunicação, o emissor utiliza PIO para preencher uma das entradas na cadeia de blocos de controle de DMA Tx DMA Requests (para quadros grandes) ou para copiar o quadro diretamente para o *buffer* circular Send Ring na memória SRAM da NIC (para quadros pequenos). A seguir, o emissor aciona o mecanismo de *Doorbell* da interface de rede com o objetivo de criar uma nova entrada na fila circular Tx FIFO Queue, sinalizando para o MCP que um novo quadro deve ser enviado. Para quadros grandes, a transferência entre Host e NIC é feita pelo controlador de DMA Host-NIC (1) e o quadro é enviado pelo MCP assim que a operação de DMA acaba (2). Quadros pequenos são enviados assim que o mecanismo de *Doorbell* é acionado, pois nesse momento o quadro já foi copiado para a memória da interface de rede.

Uma operação similar acontece durante a recepção de quadros: quando um quadro chega através da SAN, o MCP se encarrega de recebê-lo e criar uma nova entrada na Rx DMA Requests, informando ao controlador de DMA que um novo quadro deve ser transferido para a memória do host. Esse quadro é copiado assincronamente (3) para um *buffer* temporário na memória principal do host, o Staging Buffer, e fica armazenado lá até que a aplicação esteja pronta para recebê-lo. A utilização do Staging Buffer simplifica bastante o funcionamento do Núcleo Básico Myrinet pois permite que quadros recebidos sem a solicitação da aplicação sejam tratados pelo MCP, dispensando a utilização de interrupções. Além disso o Staging Buffer aumenta a flexibilidade do sistema permitindo que protocolos leves possam executar ações logo após o recebimento dos quadros e antes que o quadro seja liberado para a aplicação. Uma cópia extra, entre o Staging Buffer e o *buffer* da aplicação (4), é adicionada ao algoritmo de comunicação, um preço pequeno a se pagar se comparado com o atraso associado ao tratamento de interrupções.

É importante observar que o Núcleo Básico tira proveito do fato de que as transferências Host/NIC e NIC/SAN podem ser efetuadas concorrentemente tanto no envio quanto no recebimento de quadros para aumentar o desempenho da comunicação. A tecnologia de rede Myrinet permite que até quatro operações de DMA sejam efetuadas ao mesmo tempo: duas transferências entre a memória principal do host e a memória da NIC, uma no nodo emissor e uma no nodo receptor, e duas transferências entre a rede e a memória SRAM da NIC, novamente uma em cada um dos

nodos envolvidos na comunicação. Essa característica permite que um *pipeline* de comunicação possa ser implementado quando diversos quadros estão sendo transmitidos entre dois nodos em uma SAN Myrinet. Diversos sistemas de comunicação para redes Myrinet tiram proveito dessa característica utilizando mecanismos inteligentes de fragmentação para garantir que todos os estágios do *pipeline* são saturados durante a transfêrencia de mensagens grandes entre dois nodos [40, 41].

4.2.2 O MCP do `Núcleo Básico` Myrinet

Como visto na seção anterior, o processador da interface de rede desempenha um papel fundamental no algoritmo de comunicação do `Núcleo Básico` Myrinet, sendo responsável pela execução de grande parte das tarefas referentes ao envio e recebimento de quadros. De fato, apesar de trabalharem assincronamente, o processador do host e o processador da interface de rede Myrinet, programado através do MCP, executam o algoritmo de comunicação em conjunto. A figura 4.8 apresenta o fluxograma correspondente ao MCP utilizado pelo `Núcleo Básico` Myrinet.

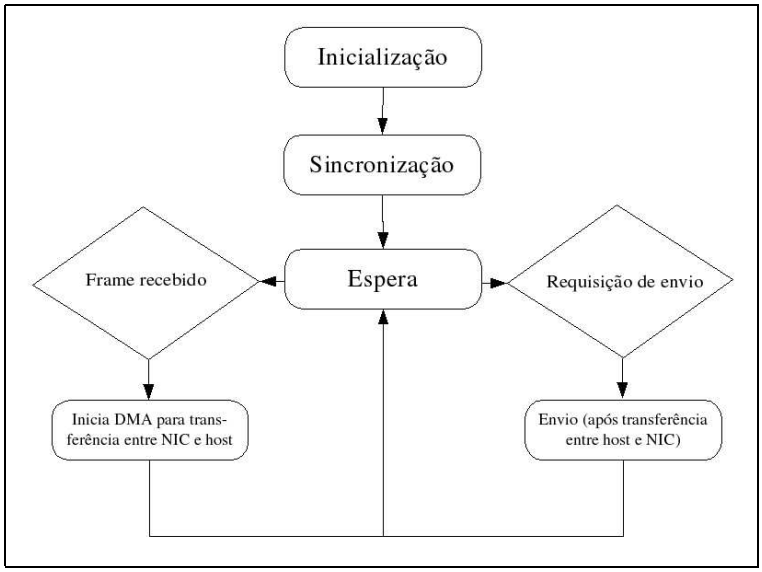

Figura 4.8: Fluxograma referente ao algoritmo implementado pelo MCP.

O MCP passa por uma etapa de inicialização, onde as informações necessárias para efetivar a comunicação são lidas da memória compartilhada e suas variáveis de controle são inicializadas, e sincronização, um *handshake* com o host que tem por objetivo assegurar que tanto o host quanto o MCP foram inicializados com sucesso e estão prontos para operar assincronamente. Em seguida, o MCP espera até que um quadro seja recebido pela interface de rede ou até que uma nova requisição de envio seja efetuada pelo host. No caso de um novo recebimento, o MCP cria a estrutura de dados utilizada pelo controlador de DMA para efetuar uma nova transferência entre a NIC e o Host e

atualiza as variáveis de controle para o próximo recebimento. No caso de uma nova requisição de envio, o MCP verifica se o DMA relacionado a esse envio, caso exista, já foi completado e, em caso afirmativo, o MCP efetua o envio. É importante observar que, devido a maneira como o algoritmo do MCP foi projetado, operações de recebimento de quadros vão sempre ter prioridade em relação as operações de envio.

Apesar de ser uma peça fundamental no algoritmo de comunicação do `Núcleo Básico` Myrinet, o MCP utilizado é extremamente simples. Diversos sistemas de comunicação projetados para a tecnologia de rede Myrinet utilizam o processador programável da interface de rede para executar tarefas complexas, tais como traduções de endereços lógicos para endereços físicos [42], *Multicast* [36] e entrega segura [43]. Entretanto, conforme observado na seção 4.1.3, a utilização de um MCP complexo pode acarretar em um impacto indesejável no desempenho do sistema de comunicação. Além disso, não faria sentido implementar serviços de comunicação na interface de rede pois a premissa básica do sistema de comunicação proposto é delegar esses serviços às `Estratégias de Comunicação`, que podem ser ativadas ou desativadas conforme os requisitos das aplicações.

É importante observar que o `Núcleo Básico` Myrinet não pode definir `Pontos de Ação` no algoritmo do MCP pois o compilador disponibilizado pelos fabricantes da tecnologia de rede Myrinet não suporta *templates*. Além disso, permitir a definição de `Pontos de Ação` no MCP aumentaria desnecessariamente a complexidade do projeto e implementação das `Estratégias de Comunicação`.

4.2.3 Estrutura do Quadro

A estrutura dos quadros utilizados pelo `Núcleo Básico` Myrinet é apresentada na figura 4.9. O campo `src` é preenchido pelo nodo emissor antes do envio com o objetivo de informar ao nodo receptor quem é o remetente do quadro. O campo `protocol header` é utilizado para armazenar toda a meta-informação que as `Estratégias de Comunicação` precisam adicionar aos quadros. Esse campo possui tamanho variável, que vai depender da configuração do protocolo composto.

4.2.4 Serviços Básicos de Comunicação

Diversos sistemas de comunicação projetados para a tecnologia de rede Myrinet assumem que o hardware é confiável e por esse motivo não implementam mecanismos de detecção de erros e

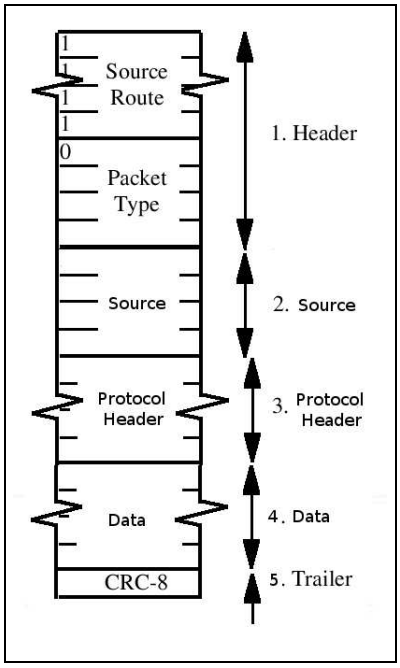

Figura 4.9: Estrutura de um quadro no `Núcleo Básico` Myrinet.

re-transmissão [19, 44]. De fato, o risco de um quadro ser perdido ou corrompido em uma rede Myrinet é tão pequeno que é seguro deixar os mecanismos que garantem entrega segura a cargo de protocolos leves projetados através de `Estratégias de Comunicação`. Prover esse serviço através de uma `Estratégia de Comunicação` tem a vantagem de permitir que aplicações que implementam mecanismos de verificação e re-transmissão possam remover o serviço do sistema de comunicação se necessário. Além disso, implementações específicas para determinadas aplicações ou tecnologias de rede podem ser desenvolvidas.

Backpressure, o mecanismo de controle de fluxo implementado em hardware pela rede Myrinet, é utilizado pelo `Núcleo Básico` para prover um mecanismo simples de controle de fluxo em nível de enlace. Mecanismos mais sofisticados devem ser providos por `Estratégias de Comunicação` projetadas para esse fim já que algumas aplicações podem implementar os seus próprios mecanismos de controle de fluxo.

Outro fator limitante do `Núcleo Básico` é que apenas a troca de mensagens ponto-a-ponto é suportada. *Multicast* e broadcast são serviços essenciais para um sistema de comunicação, especialmente em arquiteturas paralelas como agregados de PCs onde esses serviços são peças fundamentais na implementação de operações de comunicação coletivas. Protocolos-leves que disponibilizam esses serviços podem ser facilmente implementados sobre as primitivas ponto-a-ponto ou através de mecanismos mais eficazes.

É importante mencionar também que o `Núcleo Básico` não fornece mecanismos de com-

partilhamento já que muitas aplicações paralelas são executadas em ambientes dedicados. Na rede Myrinet, compartilhamento pode ser facilmente disponibilizado por uma `Estratégia de Comunicação` que utilize mecanismos como os descritos em [41], provendo uma implementação compatível com o padrão *Virtual Interface Architecture* (VIA).

4.3 `Myrinet Network` - `Núcleo Básico` Myrinet para o EPOS

Uma implementação do sistema de comunicação descrito no capítulo anterior foi desenvolvida sobre o sistema operacional EPOS, o sistema operacional orientado à aplicação utilizado no projeto SNOW [24]. O sistema de comunicação do EPOS é composto por três famílias de abstrações: `Communicator`, `Channel`, e `Network` (figura 4.10). O `Communicator` tem como membros *end-points* para a comunicação, tais como Link, Port, e Mailbox, e é utilizado como a principal interface entre o sistema de comunicação e as aplicações. Entretanto, `Communicators` não são a única interface visível para o sistema de comunicação pois a arquitetura em componentes do EPOS permite que outros componentes do sistema de comunicação sejam utilizados isoladamente, até mesmo pelas aplicações. O `Channel`, a segunda família de abstrações do sistema de comunicação do EPOS, disponibiliza componentes que garantem que dados injetados no sistema de comunicação através de um `Communicator` seja transmitida de acordo com as características desse `Communicator`. Membros da família `Channel` incluem Datagram e Stream.

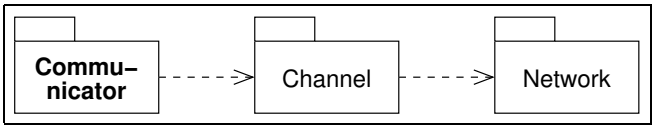

Figura 4.10: Famílias de abstrações que compõem o sistema de comunicação do EPOS [24].

`Network`, a terceira família que compõe o sistema de comunicação, é responsável por abstrair propriedades distintas de diferentes tecnologias de rede através de uma interface comum, apresentada na figura 4.11, de maneira a manter o restante do sistema de comunicação independente da arquitetura de hardware. `Networks` interagem com o hardware de comunicação através de membros da família de `Mediadores de Hardware` `NIC`.

Os conceitos explorados no capítulo anterior foram utilizados, dentro do contexto do projeto SNOW, para projetar membros da família `Network`. O núcleo de comunicação básico descrito na seção 4.2 foi adaptado e integrado ao EPOS como o membro `Myrinet_Network`

```
int send (Node_Id dst, void * buf, unsigned int len);
int receive (Node_Id * src, void * buf, unsigned int * len);
```

Figura 4.11: Interface da família `Network` do sistema de comunicação do EPOS.

da família `Network`. As chaves de configuração que ativam a utilização de protocolos leves foram implementadas como `Características de Configuração` para a `Network Myrinet` e os mecanismos de seleção, configuração e combinação estática de `Estratégias de Comunicação` foram integradas ao processo de geração de instâncias de sistemas operacionais do EPOS.

Com o objetivo de medir o desempenho do `Núcleo Básico` implementado para a tecnologia de rede Myrinet dois nodos do cluster SNOW, construído para ser utilizado no contexto do projeto de mesmo nome, foram interligados entre si e um *benchmark* Ping-Pong foi utilizado para medir a latência da comunicação entre esses nodos. O Ping-Pong consiste em aplicações que trocam mensagens entre si com o objetivo de medir o *Round Trip-Time* da comunicação, que corresponde ao dobro da latência.

A figura 4.12 apresenta uma comparação entre o desempenho do `Núcleo Básico` Myrinet sobre o sistema operacional EPOS e o GM, sistema de comunicação em nível de usuário disponibilizado pela Myricom para o sistema operacional Linux. É importante observar que, enquanto o `Núcleo Básico` Myrinet é minimalista, delegando a implementação de serviços de comunicação às `Estratégias`, o GM implementa diversos dos serviços requisitados pelas aplicações de forma monolítica dentro do sistema de comunicação. É importante observar que apenas a latência da comunicação foi avaliada pois a taxa de transmissão de ambos os sistemas de comunicação podem ser inferidas através da latência.

4.4 `Estratégias de Comunicação` Implementadas

Com o objetivo de validar o `Framework Meta-Programado` descrito no capítulo 3, e como uma primeira etapa no desenvolvimento de serviços de comunicação para as aplicações paralelas suportadas pelo SNOW, um conjunto de `Estratégias de Comunicação` genéricas foi desenvolvido. Diversas aplicações necessitam de mecanismos de fragmentação, controle de fluxo e entrega segura de mensagens e por isso `Estratégias de Comunicação` que disponibilizam esses serviços foram implementadas e o seu desempenho avaliado.

As `Estratégias de Comunicação` descritas nessa seção foram desenvolvidas e depu-

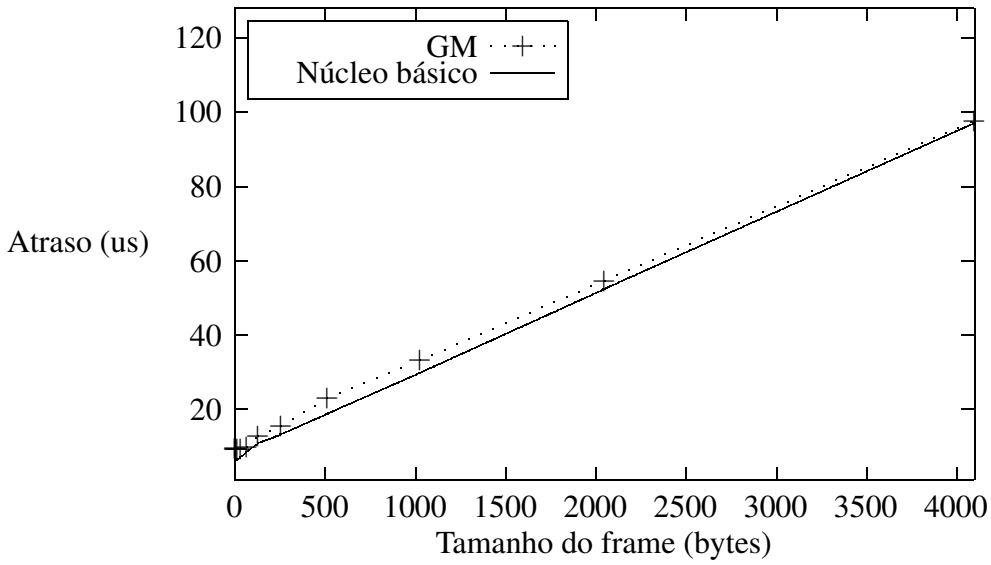

Figura 4.12: Comparação entre o desempenho do `Núcleo Básico` Myrinet/EPOS e o GM.

radas em conjunto com o `Núcleo Básico` Myrinet descrito na seção anterior. Entretanto, a implementação dessas `Estratégias` não utiliza nenhum recurso especial da tecnologia de rede Myrinet e apenas `Pontos de Ação` genéricos descritos na seção 3.1.1 for utilizados para interagir com o `Núcleo Básico` Myrinet. Portanto, os serviços de comunicação providos por essas `Estratégias` devem ser genéricos o suficiente para serem reutilizados em conjunto com `Núcleos Básicos` desenvolvidos para outras tecnologias de rede. É importante observar que implementações específicas para a tecnologia de rede Myrinet poderiam ser mais eficientes já que determinados recursos providos pelo hardware de comunicação poderiam ser utilizado na implementação dos serviços.

Fragmentação é o processo de dividir mensagens grandes em pedaços menores durante o envio de acordo com a MTU do sistema de comunicação. Esses fragmentos devem ser re-organizados pelo nodo receptor após o recebimento do último fragmento referente à mensagem e por isso informações de controle devem ser adicionadas aos quadros referentes aos diversos fragmentos da mensagem. A `Estratégia de Comunicação` que implementa o serviço de fragmentação utiliza os `Pontos de Ação` *antes do envio* e *permitir envio* para interceptar os envios de quadros realizados pelo `Núcleo Básico`, dividindo os quadros a serem enviados de acordo com a MTU definida para o sistema e utilizando o próprio `Núcleo Básico` para enviar os fragmentos. Um campo inserido no cabeçalho do primeiro fragmento de cada quadro é utilizado para informar ao nodo receptor o número de fragmentos correspondente ao quadro sendo transmitido. No nodo receptor, o `Ponto de Ação` *após o recebimento* é utilizado pela `Estratégia` e apenas depois

que o último fragmento referente ao quadro sendo transmitido é recebido é que o quadro é copiado para a memória do host. O `Núcleo Básico` Myrinet permite que a MTU seja re-configurada dinamicamente e portanto o comportamento do protocolo de fragmentação pode ser alterado em tempo de execução.

Controle de fluxo é o processo de ajustar o fluxo de dados durante a comunicação, com o objetivo de assegurar que o nodo receptor possa tratar todos os quadros enviados a ele. Existem diversos mecanismos de controle de fluxo, dentre os quais podemos destacar a comunicação *rendezvous*, onde o nodo receptor fornece ao nodo transmissor um buffer disponível para o recebimento antes que o transmissor possa enviar a mensagem, e o mecanismo de *tokens*, onde o transmissor deve possuir créditos referentes ao receptor para que possa enviar uma mensagem. O protocolo de controle de fluxo implementado para o SNOW utiliza um dos mecanismos mais comuns para comunicação assíncrona: o *xon-xoff*. Nesse mecanismo, o receptor envia uma mensagem `xoff` para o transmissor quando os seus buffers para o recebimento de mensagens estão cheios fazendo com que o transmissor aguarde até que um `xon` seja enviado pelo nodo receptor, sinalizando que esse nodo está novamente apto a receber mensagens. Os `Pontos de Ação` utilizados pela `Estratégia de Comunicação` que implementa esse serviço são o *antes do recebimento*, onde o `xoff` é enviado ao nodo emissor caso não exista espaço disponível nos *buffers* do nodo receptor, e o *após envio*, onde o nodo emissor averigua se o quadro enviado foi recebido com sucesso pelo nodo receptor. Caso o quadro não tenha sido recebido, ele é copiado em um *buffer* temporário e é re-transmitido assim que uma mensagem `xon` enviada pelo nodo receptor é recebida pelo nodo emissor. O `Ponto de Ação` *após o recebimento* é utilizado pela `Estratégia de Comunicação` para averiguar se o `xon` foi recebido.

Entrega segura é o mecanismo que garante que todas as mensagens enviadas a um nodo são recebidas, mesmo que hajam problemas na rede de comunicação. Suporte eficiente a entrega segura de mensagens é um componente fundamental de qualquer sistema de comunicação, entretanto, devido a baixa taxa de erros de transmissão na tecnologia de rede Myrinet, diversos sistemas de comunicação desenvolvidos para essa tecnologia de rede não implementam essa funcionalidade [19, 45]. O protocolo leve de entrega segura do SNOW foi implementado utilizando mecanismos de re-transmissão e *timeout*, além de estender o protocolo de controle de fluxo assegurando que as requisições de re-transmissão não sejam perdidas. O `Ponto de Ação` *apos o recebimento* é utilizado pela `Estratégia de Comunicação` para enviar uma mensagem de confirmação para o nodo emissor informando que quadro foi recebido com sucesso. Após cada envio, o `Ponto de Ação` *após envio* é utilizado para fazer com que o nodo emissor espere o recebimento da

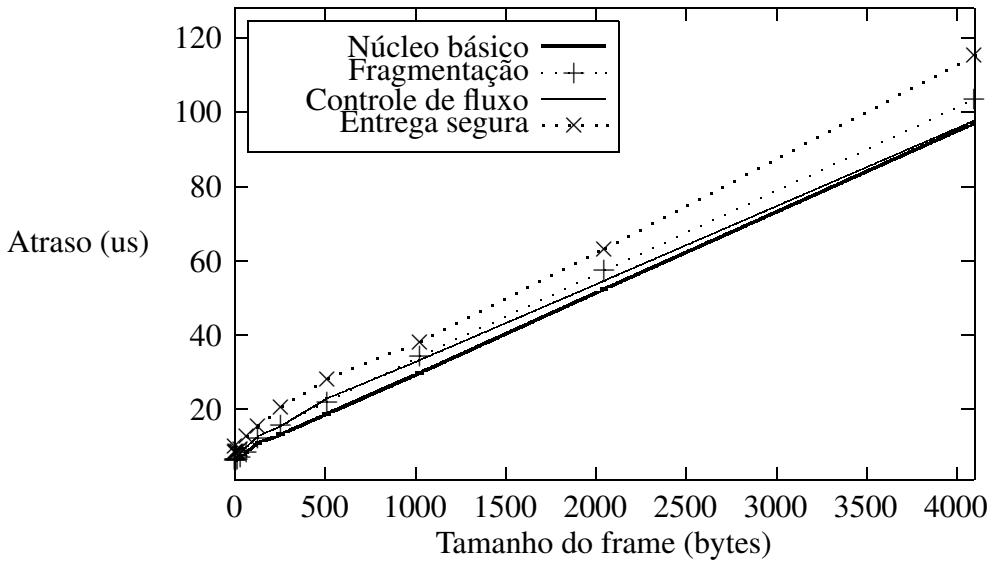

Figura 4.13: Latência do núcleo básico e dos protocolos leves implementados para o SNOW.

confirmação e re-envie o pacote caso a confirmação não chegue dentro do período especificado.

As figuras 4.13 e 4.14 exibem a latência relacionada a arquitetura básica de comunicação do sistema SNOW e aos protocolos descritos acima, individualmente e combinados. A latência foi medida utilizando o *benchmark round-trip time* para diversos tamanhos de quadro. É importante notar que o protocolo de entrega segura estende as funcionalidades do protocolo de controle de fluxo e por isso não faria sentido apresentar a latência da combinação desses dois protocolos.

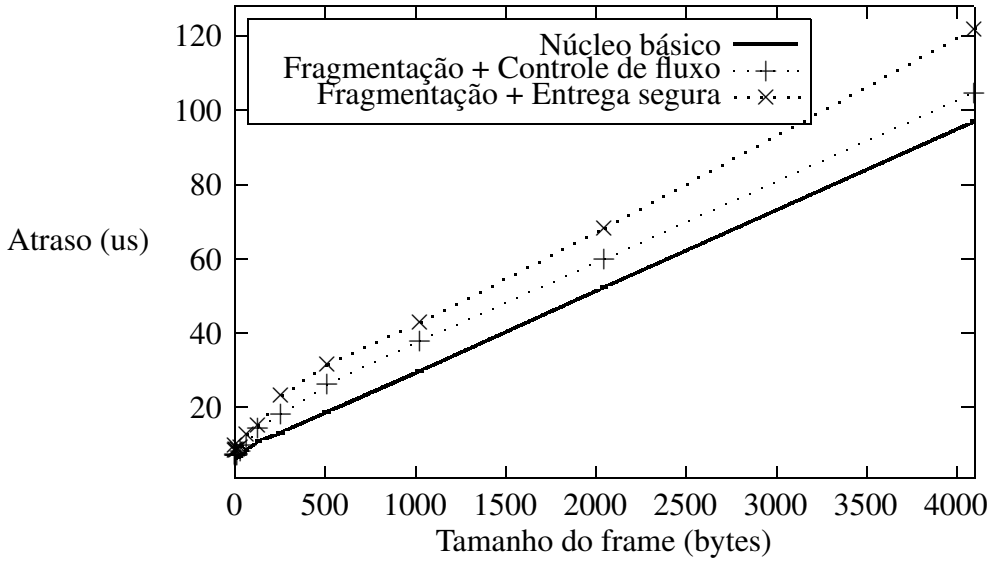

Figura 4.14: Latência do núcleo básico e das diferentes combinações de protocolos leves.

Capítulo 5

Conclusão e Trabalhos Futuros

A nova geração de sistemas computacionais e aplicações distribuídas vem motivando a academia e a indústria a propor novas arquiteturas para os sistemas de comunicação modernos. Sistemas de comunicação tradicionais provêm aos seus usuários um conjunto fixo de serviços, arquitetura que prejudica o desempenho de certas classes de aplicações e implica no desenvolvimento e utilização de camadas *middleware* que implementam os serviços não disponibilizados pelo sistema de comunicação. Além de prover mecanismos que permitam que as inovações tecnológicas disponibilizadas pelos fabricantes de redes de comunicação sejam utilizadas eficientemente, extensibilidade e configurabilidade são requisitos imprescindíveis para que os sistemas de comunicação modernos possam adaptar-se aos requisitos das aplicações.

O sistema de comunicação cuja arquitetura foi discutida nesse trabalho foi projetado com o objetivo de atender os requisitos das aplicações dedicadas suportadas pelo ambiente de programação paralela SNOW. O fato desse sistema de comunicação ser extensível permite que novos serviços de comunicação possam ser implementados sob demanda sem a utilização de camadas *middleware* e a configurabilidade do sistema permite que todos os componentes possam adaptar-se aos requisitos ditados pela aplicação. O principal componente do sistema de comunicação consiste em um `Framework Meta-Programado` que provê mecanismos que facilitam a criação, configuração e combinação dos serviços de implementados pelas `Estratégias de Comunicação`.

Diversos projetos de pesquisa se propõem a desenvolver sistemas de comunicação que utilizam protocolos modulares projetados sobre frameworks. O projeto *Tau* (Transport and up) [46] implementa composição de protocolos sem encapsulamento em camadas através de um framework dinâmico projetado para suportar protocolos fim-a-fim. O *X-Kernel Protocol Framework* [47] consiste em um conjunto de protocolos modulares que podem ser dinamicamente ligados a um fra-

mework de comunicação que provê mecanismos para que um protocolo possa invocar operações de outros protocolos, isto é receber/enviar mensagens de/para um protocolo adjacente, além de uma coleção de bibliotecas para a manipulação de mensagens, eventos e *threads*. O projeto *CANEs* (*Composable Active Network Elements*) consiste em um framework que inclui uma terminologia consistente, especificações de interface e um conjunto de requisitos funcionais para construção de *redes ativas* [48]. CANEs define um algoritmo de processamento de pacotes genérico que pode ser configurado através de instruções simples que podem ser inseridas em determinados pontos desse algoritmo.

O principal diferencial do sistema de comunicação proposto para o SNOW em relação a esses projetos de pesquisa é o fato de toda a seleção e combinação de protocolos ser efetuada estaticamente através do `Framework Meta-Programado`. Esse `Framework` faz uso de técnicas de Programação Gerativa [32] que trazem para tempo de compilação muitas das atividades relativas à configuração do sistema. Diversos aspectos do sistema de comunicação podem ser re-configurados dinamicamente, tais como as características do algoritmo de comunicação disponibilizado pelo `Núcleo Básico` e dos serviços implementados pelas `Estratégias de Comunicação`. Entretanto, a escolha de quais `Estratégias` vão estar ativas em tempo de execução é feita durante a etapa de geração do sistema.

Além da arquitetura do sistema de comunicação, as decisões de projeto referentes à implementação de alguns serviços de comunicação também foram discutidas. Como uma próxima etapa no desenvolvimento do ambiente de programação paralela SNOW, novos serviços de comunicação vão ser implementados com o objetivo de permitir que o sistema possa suportar uma gama maior de aplicações paralelas. Além disso, `Núcleos Básicos` vão ser desenvolvidos para outras tecnologias de rede de alto desempenho, permitindo que o SNOW possa ser utilizado em sistemas com configurações de hardware diferentes.

Uma etapa importante no desenvolvimento de ambientes de programação orientados à aplicação é estudar e entender melhor a relação entre as decisões de projeto referentes à utilização de recursos da infraestrutura de rede e o desempenho de comunicação das aplicações. A arquitetura do sistema de comunicação projetado para o SNOW é flexível o suficiente para permitir que diversas dessas decisões de projeto possam ser implementadas como `Estratégias de Comunicação`. O desempenho dessas `Estratégias` vai ser avaliado utilizando-se *benchmarks* mais sofisticados e aplicações reais com o objetivo de averiguar o impacto dessas decisões de projeto no desempenho de comunicação das aplicações.

A arquitetura do sistema de comunicação desenvolvido para o SNOW apresenta diversas ca-

racterísticas únicas que podem ser exploradas por implementações específicas desse sistema para domínios diversos. Como um exemplo, a utilização de implementações do sistema de comunicação para o domínio de sistemas embarcados e redes de sensores sem fio apresenta a vantagem de possibilitar que o sistema seja configurado de acordo com as características do ambiente de hardware. Diversas implementações de um determinado serviço de comunicação, com diferentes características em relação ao tamanho final (*footprint*) da `Estratégia` que implementa esse serviço, podem ser providas permitindo que a melhor implementação seja utilizada para cada configuração de hardware.

Além disso, com o objetivo de validar a hipótese de que configuração estática é suficiente para o domínio de aplicações dedicadas, o sistema de comunicação vai ser estendido de maneira a suportar a re-configuração dinâmica do protocolo de comunicação em conjunto com a configuração estática. As `Estratégias de Comunicação` já desenvolvidas vão ser utilizadas nesse novo cenário e o desempenho do novo sistema de comunicação vai ser avaliado e comparado com o desempenho atual do sistema com o objetivo de comprovar ou refutar essa hipótese.

Referências Bibliográficas

[1] M. Kaku, *Visões do Futuro - Como a ciência revolucionará o século XXI*. Rio de Janeiro, RJ - Brasil: Rocco, 2001.

[2] E. Strohmaier, J. D. M. Horst, and D. Simon, "Top500 report for june 2005," Tech. Rep., June 2005. [Online]. Available: http://www.top500.org

[3] A. A. Frohlich, P. O. A. Navaux, S. T. Kofuji, and W. Schroder-Preikschat, "Snow: a parallel programming environment for clusters of workstations," in *Proceedings of the 7th German-Brazilian Workshop on Information Technology*, Maria Farinha, Brasil, Sept. 2000.

[4] T. R. C. Santos and A. A. Frohlich, "An application-oriented communication system for clusters of workstations," in *Proceedings of the First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-1)*, Saint-Malo, França, June 2004, pp. 15–24.

[5] ——, "A customizable component for low-level communication software," in *Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*, Guelph, Canadá, May 2005, pp. 58–64.

[6] G. E. Moore, "Cramming more components onto integrated circuits," *Readings in computer architecture*, pp. 56–59, 2000.

[7] T. Suh, D. M. Blough, and H.-H. S. Lee, "Supporting cache coherence in heterogeneous multiprocessor systems," in *DATE*, 2004, pp. 1150–1157.

[8] R. A. F. Bhoedjang, *Communication Architectures for Parallel-Programming Systems*. Amsterdam, Holanda: PhD thesis, Dept. of Computer Science – Vrije Universiteit, June 2000.

[9] J. Dongarra, T. Sterling, H. Simon, and E. Strohmaier, "High-performance computing: Clusters, constellations, mpps, and future directions," pp. 51–59, Mar. 2005.

[10] G. Bell and J. Gray, "What's next in high-performance computing," *Communications of the ACM*, vol. 45, no. 2, pp. 91–95, Feb. 2002.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," in *Parallel Computing*, vol. 22, Sept. 1996, pp. 789–828.

[12] G. Burns, R. Daoud, and J. Vaigl, "Lam: An open cluster environment for mpi," in *Proceedings of the Supercomputing Symposium*, 1994, pp. 379–386.

[13] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "User-level interprocess communication for shared memory multiprocessors," in *ACM Transactions on Computer Systems*, May 1991, pp. 175–198.

[14] H. Hellwagner, W. Karl, M. Leberecht, and H. Richter, "Sci-based local-area shared-memory multiprocessor," in *Proceedings of the International Workshop on Advanced Parallel Processing Technologies - APPT'95*, Beijing, China, Sept. 1995.

[15] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson, "Glunix: A global layer unix for a network of workstations," *Software - Practice and Experience*, vol. 28, no. 9, pp. 929–961, 1998.

[16] S. Merugu, S. Bhattacharjee, E. W. Zegura, and K. L. Calvert, "Bowman: A node os for active networks," in *INFOCOM '00: Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies*, Mar. 2000, pp. 1127–1136.

[17] W. Schroder-Preikschat, *The Logical Design of Parallel Operating Systems*. Englewood Cliffs, USA: Prentice-Hall, Inc., 1994.

[18] R. H. Campbell, N. Islam, and P. Madany, "Choices, frameworks and refinement," in *Computing Systems*, 1992, pp. 217–257.

[19] L. Prylli and B. Tourancheau, "Bip: A new protocol designed for high performance networking on myrinet," in *IPPS/SPDP Workshops*, ser. Lecture Notes in Computer Science, vol. 1388, Orlando, USA, Mar. 1998, pp. 475–485.

[20] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler, "Multiprotocol active messages on a cluster of SMP's," in *Proceedings of Supercomputing'97*, San Jose, EUA, Nov. 1997.

[21] H. Tezuka, A. Hori, Y. Ishikawa, and M. Sato, “PM: An operating system coordinated high performance communication library,” in *High-Performance Computing and Networking*, ser. Lecture Notes in Computer Science, vol. 1225, Apr. 1997, pp. 708–717.

[22] D. Schmidt, D. Box, and T. Suda, “Adaptive: A flexible and adaptive transport system architecture to support lightweight protocols for multimedia applications on high-performance networks,” 1992.

[23] A. B. Maccabe, P. G. Bridges, R. Brightwell, R. Riesen, and T. Hudson, “Highly configurable operating systems for ultrascale systems,” in *Proceedings of the First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-1)*, Saint-Malo, França, June 2004, pp. 33–40.

[24] A. A. Frohlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin, Alemanha: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[25] F. V. Polpeta and A. A. Frohlich, “Hardware mediators: A portability artifact for component-based systems,” in *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, vol. 3207, Aizu, Japão, 2004, pp. 271–280.

[26] A. L. G. Sanches and A. A. Frohlich, “Epos-mpi: a highly configurable run-time system for parallel applications,” in *Proc. SBAC 2004*, Foz do Iguaçu, Brasil, Oct. 2004.

[27] M. Baker, “Cluster computing white paper,” IEEE Computer Society Task Force on Cluster Computing (TFCC), Tech. Rep. Version 2.0, Dec. 2000.

[28] G. software GmbH, “Codine: Computing in distributed network environments,” Tech. Rep., Jan. 1999.

[29] G. G. H. Cavalheiro and P. O. A. Navaux, “Dpc++: uma linguagem para processamento distribuído,” in *Proceeding os the Symposium on Computer Architecture and High-Performance computing*, vol. 2, Florianópolis, Brasil, Sept. 1993, pp. 732–744.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995.

[31] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Reading, MA: Addison-Wesley, 1997.

[32] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

[33] S. Sunder and D. R. Musser, "A metaprogramming approach to aspect oriented programming in c++," Mar. 2001.

[34] *Myrinet-on-VME Protocol Specification*, VITA Standards Organization, 1998, aNSI/VITA 26-1998. [Online]. Available: http://www.myri.com/open-specs/myri-vme-d11.pdf

[35] M. Gerla, P. Palnati, and S. Walton, "Multicasting protocols for high-speed, wormhole-routing local area networks," *SIGCOMM*, pp. 184–193, 1996.

[36] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal, "Efficient multicast on myrinet using link-level flow control," in *ICPP '98: Proceedings of the 1998 International Conference on Parallel Processing*. IEEE Computer Society, 1998.

[37] "Write combining memory implementation guidelines," Intel, Tech. Rep. 244422-001, Nov. 1998.

[38] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin, "User-space communication: A quantitative study," in *Proceedings of the IEEE/ACM SC '98 Conference*. Orlando, FL: IEEE Computer Society, Nov. 1998.

[39] *PCI64 Programmer's Documentation*, Myricom, 2001. [Online]. Available: http://www.myri.com/myrinet/PCI64/programming.html

[40] A. A. Frohlich, G. P. Tientcheu, and W. Schroder-Preikschat, "Epos and myrinet: Effective communication support for parallel applications running on clusters of commodity workstations," in *Proceedings of 8th International Conference on High Performance Computing and Networking*, Amsterdam, Holanda, May 2000, pp. 417–426.

[41] M.-S. Lee, J.-L. Yu, and S.-R. Maeng, "Pipelined implementation of the virtual interface architecture on myrinet," 2001.

[42] C. Dubnicki, A. Bilas, K. Li, and J. Philbin, "Design and implementation of virtual memory-mapped communication on myrinet, Tech. Rep. TR-570-97, 1997.

[43] R. Bhoedjang, K. Verstoep, T. Ruhl, H. E. Bal, and R. F. H. Hofman, "Evaluating design alternatives for reliable communication on high-speed networks," in *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 71–81.

[44] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal, "User-level network interface protocols," *IEEE Computer*, vol. 31, no. 11, pp. 53–60, Nov. 1998.

[45] K. Verstoep, R. A. F. Bhoedjang, T. Ruhl, H. E. Bal, and R. F. H. Hofman, "Cluster communication protocols for parallel-programming systems," Vrije Universiteit, Amsterdam, Holanda, Tech. Rep. 3, 1998.

[46] R. Kravets, K. Calvert, and K. Schwan, "Dynamically configurable communication protocols and distributed applications: Motivation and experience, Tech. Rep. GIT-CC-96-16.

[47] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, 1991.

[48] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura, "Reasoning about active network protocols," in *ICNP '98: Proceedings of the Sixth International Conference on Network Protocols*. IEEE Computer Society, 1998.