Uma Arquitetura de Implementação para Redes Ativas

Flavio Paiva Junqueira — flavio@gta.ufrj.br
Otto Carlos M. B. Duarte * — otto@gta.ufrj.br

Grupo de Teleinformática e Automação - GTA
COPPE/PEE - Programa de Engenharia Elétrica
Universidade Federal do Rio de Janeiro
Caixa postal 68504 - CEP 21945-970 - Rio de Janeiro - RJ - Brasil

Resumo

Redes ativas permitem a programação dos nós das redes, possibilitando a implantação rápida, flexível e de baixo custo de novos serviços. No modelo de programação da rede utilizado neste artigo, pacotes contendo apenas dados são substituídos por outros que carregam programas, e são denominadas cápsulas. As cápsulas são executadas a cada nó ativo pelos quais passam, onde estes nós seguem o mesmo modelo computacional.

A proposta para IP ativo utiliza o campo de opções do protocolo IPv4 e IPv6, para distinguir os pacotes IP das cápsulas IP. Baseado nesta proposta para IP ativo, apresenta-se neste artigo uma arquitetura de implementação para uma máquina de processamento que é um novo componente adicionado ao nível IP. A arquitetura proposta é baseada na utilização de máquinas virtuais Java e de um mecanismo de *garbage collection* replicante e concorrente que aumenta o desempenho em termos de execução dos programas contidos nas cápsulas.

Abstract

Active networking enables programming the network nodes, making the deployment of new services faster, flexible and cost-effective. The network programming model used in this article is the integrated one. This model uses capsules carrying data and executable code, instead of conventional packets carrying just data. The capsules are executed at each active node along their path, which follows a commom computational model.

The active IP proposal uses the options headers of both IPv4 and IPv6 to distinguish the IP packets and IP capsules. Based on this proposal, this article shows an implementation architecture to a processing engine, placed at IP level. This architecture uses the Java virtual machine to interpret the executable code and a *garbage collector* concurrent and replicating, which improves the performance of the execution in the active nodes.

*Este trabalho foi realizado com recursos da UFRJ, FUJB, PROTEM-CC, CNPq, FAPERJ, CAPES e COFECUB

1 Introdução

Nas redes de computadores tradicionais, os nós intermediários transportam *bits* de uma forma passiva, isto é, os dados do usuário são transmitidos de forma transparente e são indiferentes para os nós da rede [1]. As funções a serem executadas estão limitadas a um conjunto de opções pré-definidas presentes nos cabeçalhos das mensagens.

As redes ativas constituem um novo paradigma onde é possível a programação da rede. Os programas são injetados nos nós da rede e permitem a realização de computações sobre os pacotes que trafegam por estes nós, denominados de nós ativos. As computações realizadas sobre os pacotes podem modificar o seu conteúdo, permitindo a prestação de serviços difenciados e dependentes da aplicação [2]. A característica central que distingue máquinas programáveis (redes ativas) de máquinas configuráveis (redes passivas) é o modelo de programação utilizado. Uma máquina configurável procura estabelecer um conjunto máximo de características de alto nível que podem ser ativadas por ações atômicas e simples. Por outro lado, uma máquina programável estabelece um conjunto mínimo de primitivas as quais podem compor um número indefinido de características de alto nível [3].

As redes ativas seguem os princípios desenvolvidos primeiramente nas Redes Inteligentes [4]. As redes ativas assim como as Redes Inteligentes, permitem a implantação rápida e de baixo custo de novos serviços. Esta implantação eficiente em termos de rapidez e custo é conseguida através da separação do controle do serviço e do controle da transmissão [3, 4, 5, 6, 7].

A forma como novas computações são inseridas na rede varia entre as propostas realizadas nas pesquisas em curso atualmente para redes ativas. As principais propostas estão divididas basicamente em dois grupos. O primeiro grupo contém os modelos que propõem a programação da rede sendo feita particularmente por cada aplicação. No segundo grupo de propostas, programa-se cada nó individualmente através de pacotes contendo apenas programas, sendo a sua execução disparada por opções contidas nos cabeçalhos dos pacotes.

Dentro do primeiro grupo, a proposta de Tennenhouse *et al.* [1, 2, 8] baseia-se na substituição dos pacotes convencionais por cápsulas. Cápsulas são pequenos programas que são executados a cada nó pelos quais passam. Os dados do usuário são embutidos dentro destes programas, da mesma forma que o conteúdo de páginas de texto é embutido em um fragmento de código PostScript [1]. Tennenhouse propõe ainda a versão ativa do protocolo IP (*Internet Protocol*) [9], onde substitui-se pacotes IP por cápsulas IP . Ainda neste grupo, existe a proposta de Yemini *et al.* [3] para redes programáveis. Nesta proposta, é apresentada uma arquitetura baseada na troca de agentes entre os nós da rede que realizam processamento sobre fluxos de pacotes. Os agentes são programados utilizando a linguagem **NetScript**.

No segundo grupo de propostas encontra-se a de Zegura *et al.* [10, 11, 12]. Esta proposta baseia-se na utilização de comutadores ativos para a resolução do problema de congestionamento em redes ATM para tráfego TCP. A solução dada baseia-se em duas observações. A primeira observação revela que informações de realimentação sobre o estado do tráfego provindas da rede constituem mecanismos complexos e pouco eficientes [13]. A segunda, que pacotes TCP são divididos em um grande número de células. Se uma destas células se perde o pacote todo é inutilizado, gerando um alto grau de ineficiência para redes congestionadas [14]. Assim, em caso de congestionamento, informações contidas nas células indicam o processamento escolhido, dentro de um conjunto disponível, a ser executado sobre as células pertencentes a um determinado circuito virtual. Outra proposta dentro deste grupo é a de Smith *et al.* [4, 6] que constitui o projeto **SwitchWare**. Este projeto consiste em comutadores e pontes programáveis através de sequências de mensagens, denominadas *switchlets*, que são interpretadas ao atingir as portas de entrada do equipamento. Aplicações de gerenciamento de redes, *Firewalls* e *Web Proxies* seriam implementadas utilizando o modelo de programação único existente, ao invés de utilizar uma solução para implementação diferente para cada aplicação.

A partir da análise das principais pesquisas em redes ativas, é feito neste artigo um classificação dos principais modelo de programação para redes ativas. Neste artigo, o enfoque é a utilização de cápsulas para a programação da rede. São discutidos diversos aspectos relativos ao modelo computacional, ou seja, o conjunto de instruções para programação e os recursos disponíveis nos nós para os programas encapsulados. Baseando-se na proposta de IP ativo [9], é apresentada uma arquitetura de implementação para nós ativos baseada na linguagem Java e utilizando um mecanismo automático para reutilização da memória (*garbage collection*) otimizado.

O artigo é organizado em seis seções. Na seção 2 são apresentados os principais modelos para redes ativas e suas principais características. Os principais aspectos do modelo computacional para programação com cápsulas são apresentados na seção 3. A proposta de Tennenhouse *et al.* para IP ativo é apresentada na seção 4. A seção 5 apresenta a contribuição deste artigo que consiste em uma arquitetura de implementação para uma máquina de processamento baseada na proposta para IP ativo. Por último, na seção 6 são apresentadas as considerações finais.

2 Modelos de Redes Ativas

Os três principais modelos de redes ativas são: o baseado em agentes, o discreto e o integrado. A utilização do paradigma de agentes consiste em enviar programas em agentes para os nós ativos escolhidos que executam um processamento específico sobre um determinado fluxo. O modelo discreto separa os pacotes contendo dados dos pacotes contendo programas. Sua utilização encontra-se em comutadores/roteadores programáveis. Já o modelo integrado não faz a distinção entre os pacotes de dados do usuário e programas. Todo pacote enviado contém dados e programa. Estes três modelos serão abordados nesta seção, sendo o modelo integrado utilizado no restante do artigo.

2.1 Modelo Baseado em Agentes

O termo **agente**, utilizado neste artigo, descreve um programa que pode ser dinamicamente enviado para um sistema remoto onde será executado [3]. Programas em agentes carregam primitivas de programação que processam pacotes de um determinado fluxo e alocam recursos nos nós. O objetivo destes programas enviados sob a forma de agentes é controlar as funções exercidas nos nós da rede, possibilitando a programação de tarefas, com gerenciamento, para fluxos específicos de pacotes.

Devido à característica de envio e execução dinâmica dos agentes, questões com relação à portabilidade, à eficiência e à segurança são críticas no desenvolvimento de uma infra-estrutura para agentes [5]. A infra-estrutura para agentes deve permitir a execução de agentes em sistemas remotos de forma eficiente e independente da tecnologia utilizada no equipamento do sistema remoto. Mecanismos de autenticação devem ser criados e restrições impostas às ações permitidas nas execuções dos agentes de forma a garantir a integridade do sistema remoto.

2.2 Modelo Discreto

O processamento dos dados do usuário pode ser arquiteturalmente separado do processo de injeção de programas em um nó, com um mecanismo separado para cada função. Neste contexto, os usuários enviam pacotes contendo dados, através de nós programáveis da mesma forma que é feita atualmente. Quando o pacote é recebido, seu cabeçalho é examinado e um programa é disparado para operar sobre o conteúdo do pacote. O programa processa ativamente o pacote, possivelmente alterando o seu conteúdo. Um grau de customização é possível, pois o cabeçalho da mensagem identifica o programa que deve ser executado. Portanto, o processamento realizado sobre um determinado pacote pode ser baseado no usuário ou na aplicação [1, 4, 6, 9, 10, 11, 12].

A separação da execução e do carregamento do programa pode ser útil quando é desejável que o carregamento de programas seja cuidadosamente controlado ou quando os programas individualmente forem grandes. Esta abordagem é utilizada nas Redes Inteligentes, padronizada pelo ITU-T [7]. Na Internet, o carregamento do programa pode ser restrito ao operador do roteador, ao qual é fornecido uma "porta de fundos" através da qual sequências de código podem ser carregadas dinamicamente. Esta porta de fundos deve conceder os respectivos privilégios ao operador em questão e ainda realizar uma avaliação no código que está sendo carregado. Observa-se que permitir a operadores o carregamento dinâmico de código nos roteadores seria útil para estender as suas capacidades, mesmo que programas não realizem computações específicas sobre dados de aplicações ou usuários.

2.3 Modelo Integrado

Uma visão mais extrema de redes ativas é aquela na qual toda mensagem é um programa. Toda mensagem, neste caso denominada cápsula, que passa nos nós contém um fragmento de programa (de pelo menos uma instrução) que pode incluir dados embutidos. Quando uma cápsula chega a um nó ativo, seu conteúdo é avaliado, da mesma forma que um dispositivo Postscript interpreta o conteúdo de cada arquivo que é enviado para esta [1, 2, 8, 9].

A figura 1 fornece uma visão conceitual de como um nó ativo deve ser organizado. *Bits* chegando nos enlaces de entrada são processados por um mecanismo que identifica os limites das cápsulas, possivelmente utilizando mecanismos de enquadramento fornecido pelos protocolos tradicionais de enlace. O conteúdo da cápsula é enviado para um **ambiente de execução transiente** onde pode ser seguramente avaliado. Supõe-se que os programas são compostos de instruções primitivas, que realizam computações básicas no conteúdo da cápsula, e pode ainda chamar métodos externos, os quais podem fornecer acesso a recursos fora do ambiente transiente. A execução de uma cápsula pode resultar no escalonamento de zero ou mais cápsulas para transmissão nos enlaces de saída e podem alterar o estado não-transiente do nó. O ambiente transiente é destruído quando a avaliação da cápsula termina.

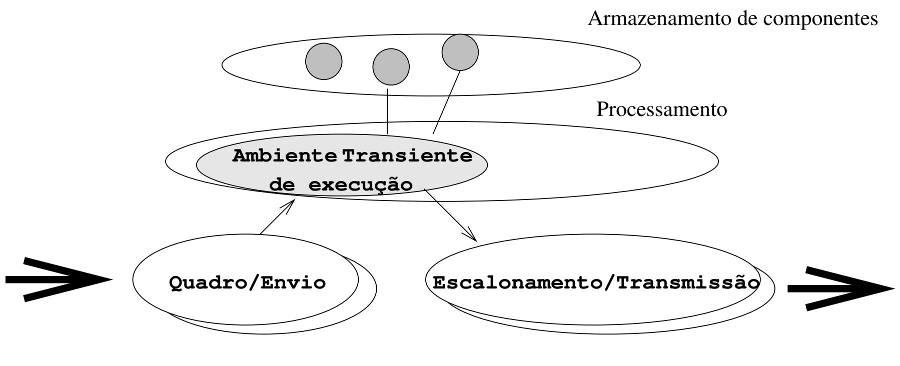

Figura 1: Organização de um nó ativo

3 Aspectos do Modelo Computacional

No modelo integrado, onde são utilizadas cápsulas no lugar de pacotes passivos, o processamento realizado no nós da rede sobre as cápsulas é baseado nos programas que estas cápsulas contém. A execução destes programas só é possível se houver um acordo com relação ao modelo computacional adotado por todos os nós ativos da rede. Neste contexto, o modelo computacional é definido como sendo o conjunto de instruções e recursos disponíveis para os programas contidos nas cápsulas [1].

Dentre os recursos disponíveis nos nós, nem todos estarão disponíveis para todos os programas encapsulados. O acesso a determinados recursos dependerá de alguma forma de autenticação, e estará disponível somente para um conjunto de sistemas fonte. O conjunto de instruções é baseado no modelo de programação adotado.

Nesta seção são apresentados os principais aspectos do modelo computacional. Primeiro são discutidos o conjunto de instruções junto com os recursos disponibilizados e em seguida são apresentadas características do modelo de gerenciamento dos recursos nos nós.

3.1 Modelo de Programação

A codificação dos programas encapsulados pode ser realizada através de código fonte, código intermediário ou código de máquina. A utilização de código fonte implica na interpretação do programa, isto é, uma conversão é realizada em tempo de execução da linguagem de alto nível para o código da máquina local. As principais vantagens deste tipo de codificação são a facilidade de implementação e a portabilidade. Por outro lado, esta conversão realizada em tempo de execução acarreta em uma degradação no desempenho. Códigos intermediários como Java são interpretados também, mas possuem a vantagem de terem, realizadas antes do tempo de execução, as fases de pré-processamento, análise léxica e checagem estática [15]. Com isto mantém-se a portabilidade aumentando-se o desempenho. O último tipo de codificação é por código de máquina, onde os programas são escritos em uma linguagem de alto nível e compilados para uma arquitetura particular de máquina. Apesar do ganho em desempenho na execução, perde-se na portabilidade.

Dentre os três tipos de codificação, o código intermediário é o mais equilibrado em termos das vantagens e desvantagens citadas. Diversas pesquisas [5] estão em andamento atualmente no sentido de aprimorar a execução para este tipo de código. A utilização de compilação *just-in-time* em tempo de execução é uma proposta para ganho em tempo de processamento durante a execução dos programas em código intermediário. Neste tipo de compilação, a geração das sequências de código de máquina, para um determinado trecho de código, é atrasada até que este trecho seja requisitado para execução. Com isto, o custo de compilação é distribuído pelo tempo de execução do programa.

Um compilador Java gera *bytecodes* a partir de um código fonte baseado na linguagem Java [16]. Os *bytecodes* são formados a partir do conjunto de instruções existentes para a máquina virtual Java, e constituem o código intermediário. O interpretador é um dos módulos presentes na máquina virtual. A máquina virtual será abordada com mais detalhes na seção 5.

A execução dos programas encapsulados ocorre segundo foi apresentado na figura 1. Para cada cápsula é criado um ambiente transiente para a sua execução. Em uma abordagem restrita, as cápsulas não teriam acesso a recursos além do ambiente transiente de execução. Embora isto seja suficiente para o propósito de diversas aplicações, algumas aplicações requerem o acesso a recursos além do ambiente transiente. Existem três formas pelas quais os programas podem acessar recursos que estão além do ambiente transiente:

- componentes de fundação: os componentes de fundação são simplesmente métodos que implementam o acesso controlado dos programas aos recursos disponíveis nos nós. Estes métodos estão disponíveis universalmente e são implementados fora do ambiente transiente de execução. Para cada cápsula, apenas um subconjunto destes métodos estará disponível, dependendo dos privilégios concedidos pelo mecanismo de autenticação. A existência destes métodos dá-se devido à necessidade de acesso a recursos como tabelas de roteamento e estado dos enlaces de transmissão para este nó;
- armazenamento ativo: ao término da execução de uma cápsula o espaço de memória ocupado pelo ambiente transiente de execução é reutilizado para dar lugar às próximas cápsulas e nenhuma informação é guardada. O espaço de armazenamento ativo é um espaço não-transiente onde uma cápsula pertencente a um fluxo pode consultar e atualizar o estado deixado por cápsulas pertencentes ao mesmo fluxo. Este estado permanece armazenado temporariamente se não houver atualização, e é conhecido como *soft state*;
- extensões: as extensões permitem a criação de novos métodos e classes nos nós para que os programas encapsulados diminuam. Desta forma, cápsulas podem carregar e deixar disponíveis métodos e classes para as execuções póstumas.

3.2 Gerenciamento dos Recursos

A interoperabilidade entre os diversos nós ativos, no que diz respeito aos programas encapsulados, é conseguida através do estabelecimento de um conjunto de abstrações dos recursos disponíveis. Todos os nós ativos devem respeitar a nomeação dada para estas abstrações, sendo as questões de eficiência e segurança problemas de implementação local.

Em termos dos recursos necessários para os programas encapsulados, supõe-se que a maior parte destes programas não necessitará de um modelo complexo de recursos. Desta forma, propõe-se em [1] um conjunto de abstrações independentes da plataforma para os recursos físicos da rede, que são: banda passante, capacidade de processamento e armazenamento transiente e não-transiente. A concessão do acesso aos recursos pode ser feita segundo classes de acesso, concedidas para conjuntos de sistemas.

A abstração do enlace deve considerar as unidades de alocação de banda e os padrões de tráfego gerados. Isto sugere que uma abordagem mais detalhada em termos de modelagem de serviço deve ser realizada para esta abstração. Em alguns ambientes, esquemas simples podem ser adotados, permitindo que cada programa encapsulado consuma uma quantidade de banda de transmissão que é proporcional ao tamanho total da cápsula.

O ambiente de execução transiente consome um armazenamento de curto prazo limitado. Duas questões importantes com relação ao armazenamento temporário são a quantidade de armazenamento entregue a uma cápsula e o intervalo de tempo em que a cápsula utiliza este espaço de armazenamento. A quantidade de armazenamento pode ser limitada por um valor padrão, o qual pode ser aumentado mediante um processo de autenticação de privilégios para uma determinada cápsula. Com relação ao tempo de utilização, supõe-se que a maioria das cápsulas executará rapidamente. No entanto, é previsto o caso onde cápsulas hibernam, e podem assim permanecer indefinidamente. A solução está no mecanismo de *garbage collection*, que reutiliza espaços de memória alocados para armazenamento transiente que permaneceram ociosos por um longo período de tempo.

Para os requisitos identificados de armazenamento de métodos externos e dados que sobrevivem a execução de um programa encapsulado, existe a abstração do armazenamento ativo. Este tipo de armazenamento pode ser dividido em dois tipos: volátil e persistente. O armazenamento volátil é utilizado para *soft states* de fluxos e componentes carregados sob demanda. Já o armazenamento persistente fornece uma abstração de longo prazo, tais como registros de eventos para propósitos de auditoria e contabilidade.

A partir da identificação dos recursos disponibilizados, pode-se particionar as atividades de gerenciamento dos recursos em três tipos:

- atribuição dinâmica: os recursos são sempre acessados por métodos externos e, portanto, o conjunto de recursos disponível é dado pelo confinamento no qual a cápsula foi inicialmente colocada. A extensão do conjunto disponível pode ser realizada através de um mecanismo de autenticação;

- validação: o código encapsulado deve ser validado para que o nó conceda o acesso aos recursos. Assume-se que um mecanismo de criptografia será a base para o mecanismo de autenticação do sistema fonte, e que todo programa a ser executado passará por um mecanismo de validação do código;

- delegação: a partir do estabelecimento do mecanismo de autenticação, percebe-se a necessidade de um mecanismo de delegação automática de autorizações de acessos a recursos, em que ambos os administradores e os implementadores concordem.

4 IP Ativo

Em [9], é feita uma proposta para extensão do mecanismo de opções do protocolo de rede IP de forma que seja possível o transporte de fragmentos de programas. O mecanismo de opções no IP foi utilizado como ponto de partida pois existe justamente para acomodar um processamento agregado que não está relacionado com o caso comum. As opções do IP tem sido utilizadas para processamento não usual de datagramas, tipicamente para monitorar e medir a rede. Como exemplo, tem-se o controle da rota desde a sua fonte, a gravação da rota percorrida e a informação agregada de temporização. A figura 2 apresenta o formato do campo de opções para o IPv4 e a figura 3 o formato do campo de opções para o IPv6.

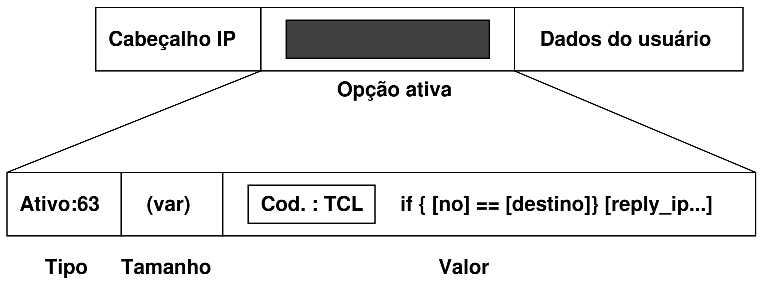

Figura 2: Formato do campo de opções para o IP ativo

A proposta para o IP ativo fornece um mecanismo para embutir fragmentos de programas em um datagrama IP, transformando-o em uma cápsula. Estes fragmentos são inseridos nos campos de opções apresentados nas figuras 2 e 3. Assim, os fragmentos são executados por nós ativos ao longo do caminho seguido pelos datagramas. Conceitualmente, a opção ativa transforma as opções do IP de um conjunto padronizado e fixo de rotinas em uma coleção extensível e integrada por uma linguagem de programação.

Para o IPv6 [17], propõe-se a utilização dos cabeçalhos de extensão de opções salto-a-salto (*Hop-by-hop options header*) e o de opções ao destino tipo 1 (*Destination options header*) em conjunto com a opção ativa. O primeiro cabeçalho permite que todos os nós executem o fragmento de código que é carregado pelo datagrama. O segundo será avaliado somente pelos nós identificados pelo cabeçalho de extensão de rota explícita (*source routing*), que também deve fazer parte do datagrama. Os cabeçalhos são identificados pelos valores 0 (opção salto-a-salto) e 59 (opção ao destino) nos campos *Next Header* do cabeçalho anterior.

Para o IPv4, dois códigos específicos são definidos para opções os quais aderem ao formato genérico tipo-tamanho-valor das opções. Estas opções são correspondentes às utilizadas pelo IPv6 ativo. A primeira é utilizada para transportar fragmentos de programas, os quais podem ser codificados em diversas linguagens. A segunda opção é utilizada para buscar nós ativos para

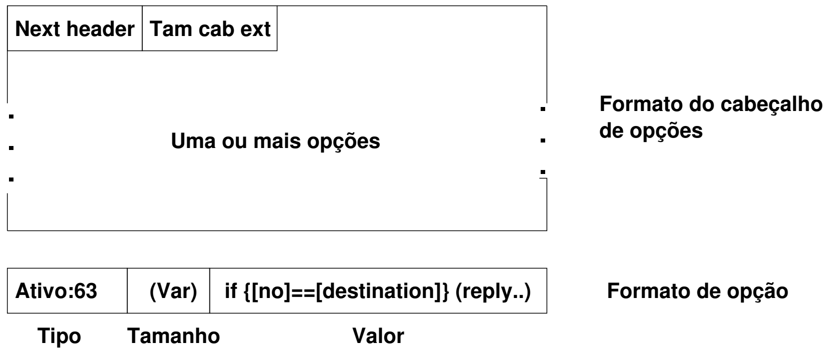

Figura 3: Formato dos cabeçalhos de opções para IPv6

as linguagens que suporta. Para fins de compatibilidade, a proposta baseia-se no requerimento apresentado na RFC 1122 [18] que diz que um nó deve ignorar silenciosamente a qualquer opção que porventura venha a ser desconhecida.

O modelo de processamento para a opção ativa é mostrado na figura 4. A **máquina de processamento**, mostrada como um retângulo denominado AP, é adjacente a camada IP, e é invocada de dentro desta quando uma cápsula chega ao nó. Desta forma, a opção ativa não forma um protocolo diferente, e portanto seu processamento não precisa seguir um padrão ponto-a-ponto. O processamento ocorre na fonte, em cada roteador ativo pelo qual o pacote passar, e no destino.

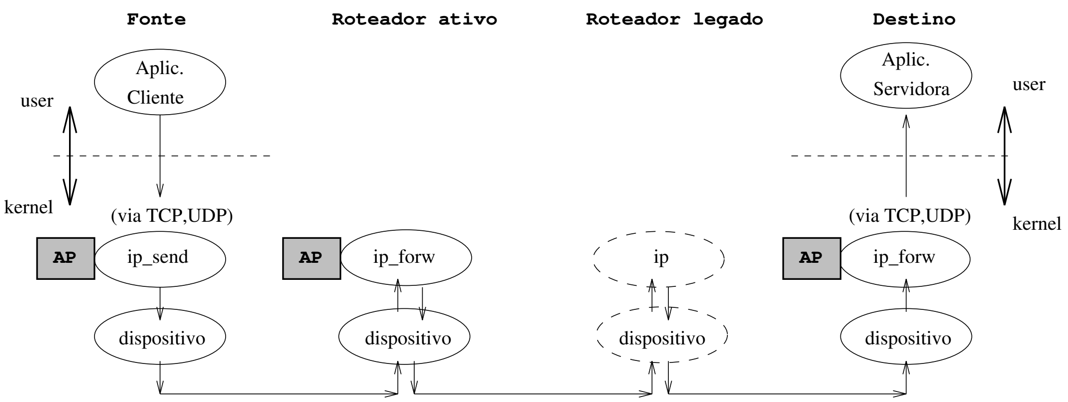

Figura 4: Processamento nos nós em uma rede ativa

A utilização de opções ativas para o IP permite uma grande flexibilidade na realização das funções de roteamento e cópia. O ambiente de processamento permite que as condições da rede sejam examinadas, o datagrama corrente seja despachado para transmissão, e datagramas adicionais sejam construídos e lançados. O cabeçalho e a carga útil do datagrama podem ser lidos e alterados. Além disto, o *soft state* do nó pode ser alterado.

Baseando-se na linguagem Tcl, alguns *scripts* são apresentados como exemplo em [9]. Na figura 5 vemos alguns destes exemplos. Os fragmentos de programas **record route** e **ping** correspondem à opção **record route** do IP e as funções ICMP de pedido/resposta, respectivamente. Outro fragmento de programa interessante é o **MTU**, que armazena o MTU de cada enlace pelo qual passa. Poderia ser utilizado para descobrir o máximo MTU em um determinado caminho para IPv6, já que a sua especificação deste protocolo sugere que a fragmentação seja feita na fonte [17]. Este mecanismo substitui o que atualmente é proposto na RFC 1981 [19].

```
record route:
    append_data 0 [node]

ping:
    if{[node] == [destination]} then {
        reply_ip [data 0]
        discard
      }
```

```
mtu:
   append_data 0 [mtu]
   if {[node] == [destination]} then {
       reply_ip [data 0]
       discard
   }
```

Figura 5: Programas de exemplo

5 Arquitetura de Implementação para a Máquina de Processamento

A arquitetura de implementação apresentada neste artigo está baseada na proposta de Tennenhouse *et al.* onde utilizam-se cápsulas no lugar de pacotes e uma máquina de processamento que é invocada pela camada IP. Esta máquina de processamento se baseia na máquina virtual da linguagem Java [16].

Nesta seção são apresentados os componentes principais da arquitetura de implementação da máquina de processamento. Apresenta-se o modelo básico da máquina virtual Java e o seu funcionamento. A máquina virtual sugerida utiliza a técnica de compilação *Just-in-Time* para otimizar a execução dos programas. O mecanismo de *garbage collection* proposto para esta máquina de processamento é apresentado em seguida, onde sugere-se o mecanismo proposto por Nettles *et al.* [20]. Esta proposta consiste em um mecanismo replicante que aumenta o desempenho geral do sistema, pois permite que processos em execução acessem a *heap* concorrentemente com o coletor. Como foi visto na seção 3, que caracteriza o modelo computacional, as instruções primitivas, os recursos dos nós disponíveis e os métodos para acesso destes recursos são definidos para a execução dos programas encapsulados. Os recursos computacionais são definidos na subseção seguinte. Por último é apresentado o diagrama com os principais módulos que compõem esta arquitetura, e as suas funcionalidades.

5.1 A Máquina Virtual Java

A linguagem Java se propõe originalmente a ser de propósito geral, orientada a objeto e semelhante a C e C++ [16]. Diferente de C e C++, Java não utiliza aritmética de ponteiros, o que torna os programas codificados nesta linguagem mais robustos e seguros. A compilação de um programa com o código fonte em linguagem Java gera *bytecodes*, colocando-os em arquivos com formato *class*, que serão interpretados pela máquina virtual Java. Os *bytecodes* são instruções para a máquina virtual Java.

Os *bytecodes* consistem no que é chamado de código intermediário. A intenção ao gerar o código desta forma é aproximar ao máximo o código gerado pela compilação do código para uma máquina específica [1]. Com isto são eliminadas as fases de pre-processamento análise léxica (*parsing*) e checagem estática (análise para detecção de possíveis erros de programação) [5, 15]. A execução do código intermediário pode ser feita através de interpretação ou através de uma nova compilação para gerar o código de máquina. No segundo caso, a nova compilação que é realizada é menos custosa em termos de processamento, pois como foi comentado, algumas fases já foram eliminadas. Desta forma, a vantagem do código intermediário sobre o código de máquina é a portabilidade. A vantagem do código intermediário sobre o código interpretado é o custo em termos de processamento.

A interpretação dos *bytecodes* pela máquina virtual pode ser realizada de duas formas: interpretação direta ou compilação *just-in-time*. Utilizando-se compilação *just-in-time* a geração de um trecho de código é atrasada até o momento de sua execução. Isto espalha o custo de compilação durante a execução do programa [5]. Para a máquina virtual a ser utilizada pela máquina de processamento proposta neste artigo sugere-se a utilização da técnica de compilação *just-in-time*. Atualmente existem diversas técnicas para o desenvolvimento de compiladores com baixo custo de processamento que podem ser utilizadas para a implementação de compiladores JIT.

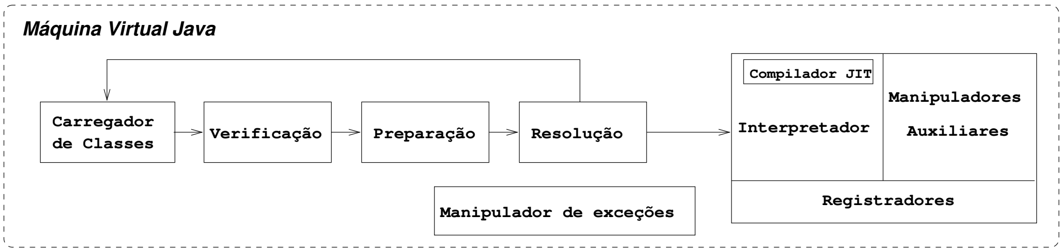

Figura 6: Máquina Virtual Java

Antes da interpretação dos *bytecodes* de uma classe, diversas fases são realizadas pela máquina virtual. Na figura 6 é apresentado um diagrama para a máquina virtual Java, obedecendo a especificação em [16]. Cada módulo na figura representa uma fase do processo de execução de um programa. O primeiro passo realizado pela máquina virtual consiste no carregamento do arquivo de classe que contém o método `main`, pelo **Carregador de Classes**. Uma vez carregada, a classe passa pelo processo de **Verificação** onde a corretude da estrutura da classe é confirmada. O processo de verificação consiste em conferir se todos os códigos de operação (opcodes) são válidos, se instruções de desvio saltam para o início de uma instrução ao invés de saltar para o meio etc. Na fase de **Preparação** seguinte todos os campos estáticos são criados e inicializados para seus valores *default*. A fase de **Resolução** resolve as referências simbólicas que correspondem à chamadas de métodos de outras classes, atribuições a atributos de outras classes etc. Se a classe correspondente à referência simbólica não houver sido carregada, uma solicitação será feita para o Carregador de Classes para que o faça. As fases de verificação, preparação e resolução fazem parte da processo de ligação (*linking*) [16].

A última fase de operação da máquina virtual consiste na interpretação. O programa em execução pode ser composto de múltiplos *threads* (linhas de execução), sendo alocado um espaço de pilha para cada um. O interpretador utiliza os registradores da máquina virtual para manipular a pilha, os *frames* e o código em execução. *Frames* são criados quando métodos são chamados e consistem em regiões de memória que são utilizadas durante a execução destes métodos para variáveis locais e resultados parciais. A região de memória utilizada por um *frame* se encontra dentro da pilha do *thread* que o utiliza. O interpretador se utiliza de **manipuladores auxiliares** que são o manipulador de *frames* que gerencia a sua criação (a destruição é feita pelo mecanismo de *garbage collection*), e o manipulador de métodos nativos (métodos existentes em bibliotecas na própria máquina local). Observa-se ainda que o módulo que realiza a compilação *just-in-time* está no interpretador. O último módulo a ser comentado é o **Manipulador de Exceções**. Este módulo trata o lançamento de exceções pelos diversos módulos da máquina virtual.

A máquina virtual utilizada para redes ativas não precisa, a princípio, considerar aplicações com múltiplos *threads*. Assim cada aplicação possui apenas uma única pilha para seus *frames*. Ainda mais, cada execução exigirá sua própria máquina virtual e terá seu próprio espaço na *heap* para instâncias de classes e para as cadeias de dados (*arrays*).

5.2 *Garbage Collection* Concorrente e Replicante

O uso de um mecanismo de *garbage collection* é necessário para garantir a alocação dinâmica segura e eficiente. Linguagens de programação como C e C++ possuem aritmética de ponteiros e permitem a alocação dinâmica de memória. Estas características tornam estas linguagens flexíveis, mas por outro lado são a causa da maioria dos problemas de quebra de programas (*crash*). O mecanismo de *garbage collection* realiza, junto com a máquina virtual, a alocação e dealocação de memória automaticamente. Esta medida impede erros propositais ou causados por imprudência do programador [16]. Erros deste tipo não seriam convenientes em nós da rede. Por isso utiliza-se *garbage collection* para gerência automática de armazenamento.

O mecanismo de *garbage collection* usual(*stop-and-copy*) varre, periodicamente ou a partir de uma exceção gerada por falta de memória, a *heap* em busca de objetos que não são mais referenciados e portanto podem ser coletados. A desvantagem deste mecanismo é que durante a ação de busca e coleta os processos em execução são interrompidos. Estas interrupções na execução dos processos, segundo medidas realizadas por Nettles *et al.*, degradam o desempenho do sistema.

A especificação da máquina virtual Java não referencia nenhum mecanismo de *garbage collection* em particular. Desta forma, para a arquitetura de implementação da máquina de processamento sugere-se a utilização do mecanismo proposto por Nettles *et al.* em [20]. Este mecanismo de *garbage collection* realiza a coleta de objetos não referenciados de forma concorrente e replicante, diminuindo os tempos de pausa em relação ao mecanismo *stop-and-copy*. De outra forma, é permitido aos processos a leitura e a escrita na *heap* enquanto o coletor concorrentemente replica os objetos para criar uma nova *heap*.

A interface para o acesso da *heap* é constituída de operações de leitura, escrita e alocação, e ainda uma raíz. A raiz contém as referências existentes para os objetos na *heap*. Como nenhuma operação de dealocação é exposta aos processos, objetos que podem ser acessados através das referências da raíz são considerados vivos e são preservados pelo sistema. A interface é apresentada na figura 7.

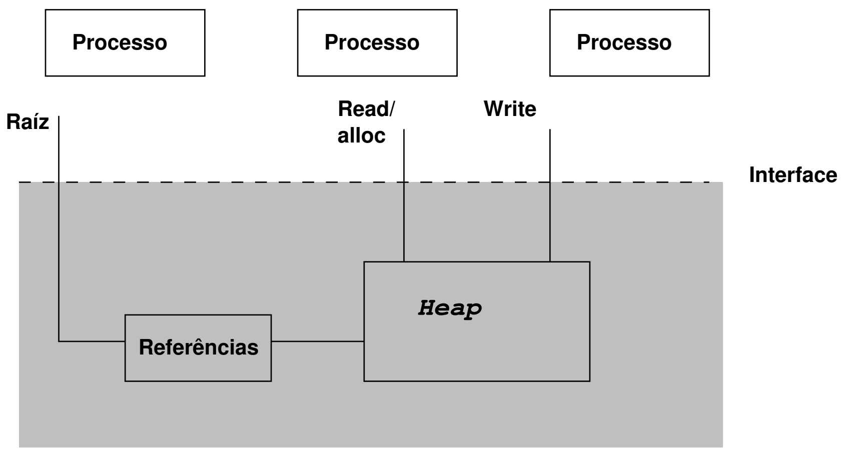

Figura 7: Interface do mecanismo de *garbage collection*

A técnica de *garbage collection* replicante consiste em copiar os objetos de forma não destrutiva. Como pode ser visto na figura 8, o mecanismo de coleção replicante opera com dois espaços de memória: o espaço ativo e o espaço replicado. Concorrentemente com os processos em execução, o coletor copia os objetos do espaço ativo para o espaço replicado. Os objetos que não possuem mais referências não são copiados. Por cópia concorrente de objetos entende-se que os processos podem continuar a realizar operações no espaço ativo. Devido a esta concorrência, algum processo pode alterar um objeto já copiado. A solução para isto é o *Redo Log* que mantém informação dos objetos que já foram copiados e foram alterados pelos processos. A única parada no sistema ocorre quando o espaço réplica é idêntico ao espaço ativo, exceto os objetos marcados no *Redo Log*. O sistema atualiza os objetos necessários e troca os papéis do espaço ativo e do espaço réplica. O tempo de parada deste mecanismo, segundo experimentos realizados por Nettles *et al.*, é bastante inferior ao do mecanismo convencional de *stop-and-copy*.

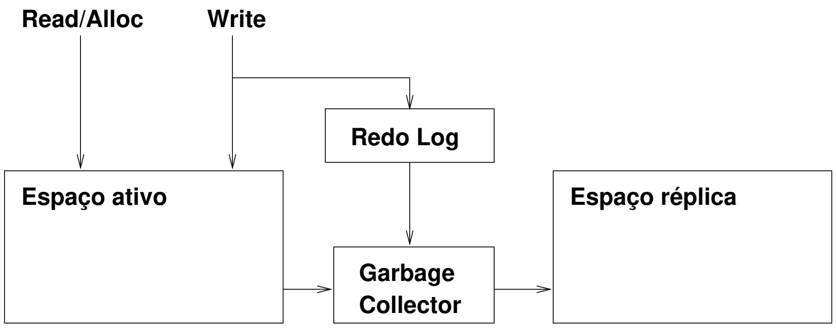

Figura 8: Coletor Replicante e Concorrente

5.3 A Arquitetura de Implementação

Baseado na proposta de Tennenhouse *et al.* para IP Ativo [9], a máquina de processamento está presente no mesmo nível da camada IP e atua em conjunto com esta camada. Programas escritos em linguagem Java estariam encapsulados no campo de opções do cabeçalho IP. A arquitetura para implementação da máquina de processamento proposta é constituída principalmente de um **Núcleo de Máquinas Virtuais** (NMV) e de módulos auxiliares que junto com o núcleo cumprem os aspectos apresentados na seção 3 para a obtenção do modelo computacional seguro, robusto e eficiente. O diagrama da arquitetura de implementação proposta é apresentado na figura 9.

O Núcleo de Máquinas Virtuais gerencia a abertura de máquinas virtuais baseadas em Java para as cápsulas que lhe são entregues. Cada cápsula terá uma máquina virtual aberta para a sua execução. Os recursos disponibilizados para a cápsula na criação da máquina virtual vão depender da identificação da origem da cápsula. Assume-se que a identificação será feita utilizando algum mecanismo de criptografia [1]. A autenticação é realizada por um **Gerenciador de Autenticações** (GA), que mantém informações sobre os privilégios que os sistemas-fonte

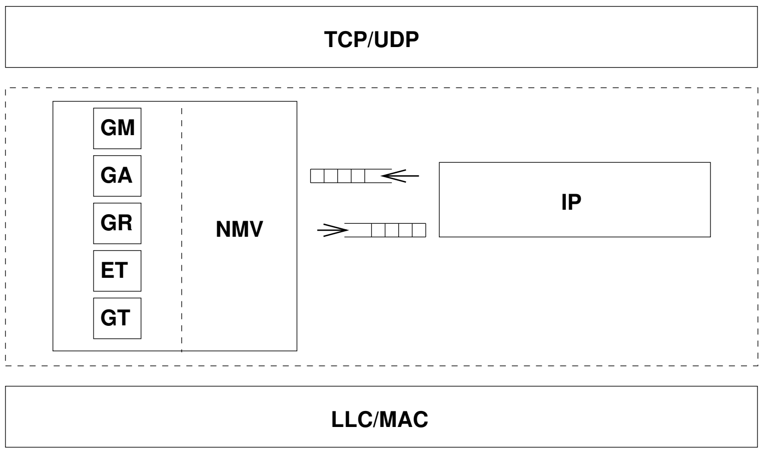

Figura 9: Arquitetura de Implementação

tem em termos de execução.

Em conjunto com o GA, o **Gerenciador de Recursos** (GR) provê o acesso propriamente dito. Todo pedido de acesso a um recurso especial do nó como acesso a tabelas de roteamento ou armazenamento não-transiente, deve ter os privilégios conferidos junto ao GA para ser realizado.

Os três últimos módulos foram baseados na arquitetura de implementação para sistemas de comunicação de alto desempenho GTA/MM [21]. Esta arquitetura citada possui três módulos básicos que são: o **Gerenciador de Memória** (GM), o **Gerenciador de Temporizações** (GT) e o **Escalonador de Tarefas** (ET). O GM é baseado no mecanismo de *garbage collection* replicante e concorrente apresentado anteriormente. O acesso é promovido segundo as operações básicas definidas para este mecanismo. O GT fornece temporizadores para ações como invalidar cápsulas que foram postas para "dormir" e assim permaneceram indefinidamente. Por último, o ET realiza a alternância entre os processos em execução na máquina de processamento.

A comunicação entre a camada IP e a máquina de processamento é feita de forma assíncrona através de filas. A fila de entrada recebe cápsulas que serão executadas por máquinas virtuais conferidas pelo núcleo NMV. A princípio existe somente uma fila de saída da máquina de processamento, mas também seria possível a existência de várias filas representando prioridades para promover qualidade de serviço.

6 Considerações Finais

As pesquisas em redes ativas destinam-se basicamente a tornar programáveis as redes de computadores. O principal objetivo da programação das redes é permitir uma implantação de novos serviços mais rápida e de baixo custo. As redes podem ser programadas através de agentes que são enviados para nós específicos, onde é executado o programa que carregam sobre um fluxo de pacotes particular. Outra forma de programar a rede é através do envio de sequências de pacotes contendo código executável para comutadores programáveis. Mas o modelo de programação adotado neste artigo foi o integrado, que se baseia na substituição de pacotes passivos por cápsulas.

O modelo computacional para cápsulas é constituído do conjunto de instruções disponíveis para programação e dos recursos disponíveis para utilização pelos programas encapsulados. A linguagem Java foi escolhida para compor o modelo computacional por gerar um código intermediário durante a sua compilação, aumentando o desempenho em tempo de execução. Para cada cápsula é concedido um espaço transiente para execução do seu programa. O conjunto de recursos do nó disponibilizado junto com o ambiente transiente depende de um mecanismo de autenticação. Dentre os vários recursos disponíveis nos nós estão o armazenamento ativo (não-transiente) e o acesso a tabelas de roteamento.

A proposta para IP ativo é baseada no modelo de cápsulas. Esta proposta constitui a base para a arquitetura de implementação para a máquina de processamento apresentada neste trabalho. A execução das cápsulas, nesta arquitetura, é realizada em máquinas virtuais Java. A existência de várias máquinas virtuais em um determinado instante de tempo é gerenciada pelo núcleo de múltiplas máquinas virtuais (NMV). Para evitar desperdício de memória e possíveis quebras do sistema devido a erros de programação é utilizado um mecanismo de *garbage collection* replicante e concorrente. Este mecanismo em particular tem a característica de não interromper os processos em execução durante a coleta. Observa-se ainda que o núcleo de máquinas virtuais é auxiliado por diversos módulos no gerenciamento do sistema.

Para trabalhos futuros propõe-se que esta arquitetura seja implementada com uma versão da máquina virtual simplificada e com um conjunto de instruções menor do que o original para a linguagem Java, para que o tempo de processamento de cada cápsula diminua.

Referências

[1] D. Tennenhouse e D. Wetherall, "Towards an Active Network Architecture", *Computer Communication Review*, vol. 26, no. 2, abril 1996.

[2] D. Tennenhouse et al, "A Survey of Active Network Research", *IEEE Communications Magazine*, pp. 80–86, janeiro 1997.

[3] T. Yemini e S. da Silva, "Towards Programmable Networks", in *IFIP/IEEE International Workshop on Distributed Systems*, outubro 1996.

[4] J. Smith et al., "SwitchWare: Accelerating Network Evolution", tech. rep., Universidade da Pensilvânia, junho 1996.

[5] J. Hartman et al., "Liquid Software: A New Paradigm for Networked Systems", Tech. Rep. 96-11, University of Arizona, 1996.

[6] S. Alexander et al., "Active Bridging", in *SIGCOMM 97*, 1997.

[7] G. Lauer, ""IN Architectures for Implementing Universal Personal Telecommunications"", *IEEE Network*, pp. 6–16, abril 1994.

[8] D. Tennenhouse et al, "From Internet to ActiveNet", *Request For Comments*, janeiro 1996.

[9] D. Wetherall e D. Tennenhouse, "The ACTIVE IP Option", in *Proceedings of the 7th SIGOPS Workshop*, Connemara, Irlanda, setembro de 1996.

[10] S. Bhattacharjee, K. Calvert e E. W. Zegura, "On Active Networking and Congestion", tech. rep., GIT-CC-96-02, College of Computing, Georgia Tech, 1996.

[11] S. Bhattacharjee, K. Calvert e E. W. Zegura, "Implementation of an Active Networking Architecture", in *Workshop Gigabit Switch Technology, Universidade de Washington, St. Louis*, julho 1996.

[12] S. Bhattacharjee, K. Calvert e E. W. Zegura, "An Architecture for Active Networking", in *High Performance Networking, White Plains, NY*, abril 1997.

[13] R. Jain et al., "Source Behavior for ATM ABR Traffic Management: An Explanation", *IEEE Communications*, pp. 50–57, novembro 1996.

[14] A. Romanov e S. Floyd, "Dynamics of TCP Traffic over ATM Networks", *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 4, pp. 633–641, maio 1995.

[15] A. Aho, R. Sethi e J. Ullman, *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[16] Sun Microsystems Inc., *The Java Virtual Machine Especification*, 1995.

[17] W. Stallings, "IPv6: The New Internet Protocol", *IEEE Communications Magazine*, pp. 96–108, julho 1996.

[18] R. Braden, "Requirements for Internet hosts - Communication Layers". Request for Comments 1122, outubro 1989.

[19] J. McCann e J. Mogul, "Path MTU Discovery for IP version 6". Request for Comments 1981, agosto 1996.

[20] J. O'Toole, S. Nettles e D. Gifford, "Concurrent Compacting Garbage Collection of a Persistent Heap", in *Proceedings, 14th Symp. Operating Systems Principles*, dezembro 1993.

[21] C. Albuquerque, "Arquitetura de Implementação de Alto Desempenho para Sistemas Multimídia", Tese de Mestrado, COPPE/UFRJ, abril 1995.