Estudo e Experimentação de uma Linguagem de Modelagem de Sistemas Baseada em Redes de Petri e Orientação a Objetos

Edna Dias Canedo

Dissertação submetida à Coordenação de Pós-Graduação em Informática da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Área de Pesquisa: Sistemas de Software

Jorge César Abrantes de Figueiredo

Orientador

Campina Grande, Paraíba, Brasil

CANEDO, Edna Dias Canedo

C221E

Estudo e Experimentação de uma Linguagem de Modelagem de Sistemas Baseada em Redes de Petri e Orientação a Objetos

Dissertação de Mestrado, Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, PB, Agosto de 2002.

141p. Il.

Orientador: Jorge César Abrantes de Figueiredo

1. Engenharia de Software
2. Redes de Petri
3. Orientação a Objetos

CDU – 519.683

"ESTUDO E EXPERIMENTAÇÃO DE UMA LINGUAGEM DE MODELAGEM DE SISTEMAS BASEADA EM REDES DE PETRI E ORIENTAÇÃO A OBJETOS"

EDNA DIAS CANEDO

DISSERTAÇÃO APROVADA EM 29.08.2002

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Orientador

PROFª PATRÍCIA DUARTE DE LIMA MACHADO, Ph.D
Examinadora

PROF. ARTURO HERNÁNDEZ DOMÍNGUEZ, Dr.
Examinador

CAMPINA GRANDE – PB

Resumo

A integração da teoria de redes de Petri com os conceitos da Orientação Objetos surgiu como uma solução para a decomposição e estruturação de modelos em Redes de Petri. Neste trabalho é apresentado um estudo e experimentação de uma linguagem de modelagem baseada em redes de Petri e Orientação a Objetos (RPOO). RPOO foi definida integrando esses dois formalismos em uma perspectiva ortogonal, permitindo que o sistema modelado tenha duas visões: uma visão do ponto de vista de redes de Petri e uma visão do ponto de vista da Orientação a Objetos. O estudo e experimentação da notação RPOO é efetuado através da sua aplicação na modelagem de sistemas de software real. Para tanto, foram desenvolvidos alguns experimentos de modelagem. Assim, os experimentos desenvolvidos serão utilizados na validação da linguagem RPOO como alternativa para a estruturação e decomposição de modelos em redes de Petri, quando da modelagem de sistemas de software distribuídos e concorrentes.

Abstract

The integration of Petri nets theory and object oriented-object concepts has emerged as a solution to decompose and structure Petri net models. In this work it is presented a study and experimentation of a modeling language based on Petri nets and oriented-object concepts (RPOO). RPOO was defined by integrating these two formalisms on an orthogonal perspective, allowing that the modeled system has two visions: one Petri net vision and one OO vision. The study and experimentation of RPOO notation is accomplished through its application on the modeling of real software systems. Some experiments of modeling had been developed. These experiments will be used to validate RPOO as an alternative for the structure and decomposition of Petri nets models, when modeling distributed and concurrent software systems.

Agradecimentos

Gostaria de agradecer, em primeiro lugar, ao meu filho Breiner Gabriel pelo completo apoio e compreensão de ficarmos distantes durante o período em que estive dedicando-me a este trabalho. O seu carinho foi fundamental para a realização desta dissertação. Obrigado a minha mãe e a minha irmã Iraides.

Ao Auro, por ter cuidado do nosso filho durante a minha ausência.

Agradeço ao meu orientador pela confiança, paciência e pela excelente orientação prestada. Obrigado por ter aceitado a orientar-me e pelo empenho.

A Dalton, pela paciência e pelos esclarecimentos.

Aos colegas do grupo de Redes de Petri, em especial a Érica e Leandro pelas contribuições, as quais foram muito importantes para a conclusão deste.

Aos colegas do mestrado.

Aos professores e funcionários do DSC.

A Aninha, por ser uma pessoa tão prestativa e carinhosa.

Finalmente, gostaria de agradecer ao Airon pela ajuda e incentivo nos momentos em que pensava não suportar mais a saudade do meu filho e as dificuldades encontradas.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação e Contexto

Desde o trabalho proposto por Carl Adam Petri no início dos anos 60, a teoria de redes de Petri tem sido extensamente estudada. Muitos pesquisadores direcionaram esforços para definir extensões, teorias de prova, métodos de análise e na construção de ferramentas para suportar a aplicação de redes de Petri [Mur89]. Dentre as extensões definidas temos: as extensões relacionadas com a capacidade de modelagem funcional—redes de Petri de alto nível, e as extensões relacionadas com a caracterização de restrições temporais—redes de Petri temporais [MAM95]. As redes de Petri de alto nível permitem uma representação mais compacta do sistema modelado, reduzindo o tamanho dos modelos. As redes de Petri coloridas (CP-Net) [Jen92c] são as redes de alto-nível mais conhecidas. As redes de Petri temporais permitem a representação das propriedades temporais de um sistema. Os métodos de análise das redes de Petri possuem notação matemática, fornecendo resultados capazes de validar a construção dos modelos, identificando inconsistências, ambigüidades e erros mesmo antes da implementação do sistema.

Embora as redes de Petri de alto nível propiciem a redução do tamanho dos modelos, para tratar da complexidade inerente dos sistemas reais é necessário mecanismos adicionais de estruturação e abstração. No caso das redes de Petri coloridas, o mecanismo adotado é a utilização de hierarquia para estruturar os modelos, resultando nas redes de Petri coloridas hierárquicas (HCPN) [Jen92c]. HCPN permite ao modelador construir modelos grandes combinando um número de pequenas redes em uma grande rede. Embora as HCPN sejam

apresentadas como solução para a estruturação, a modelagem de sistemas complexos requer facilidades de abstração que vão além das proporcionadas por HCPN [CKR94], que é a utilização da hierarquia como mecanismo de estruturação e decomposição dos modelos. A hierarquia permite apenas uma estruturação visual do modelo, não provendo um mecanismo efetivo para distribuir as responsabilidades entre os módulos. Assim, não existe uma relação contratual entre os módulos do modelo. Uma outra solução proposta para resolver o problema da estruturação de sistemas é a integração dos conceitos de orientação objeto (OO) na teoria de redes de Petri[Lak95].

A modelagem orientada a objetos disponibiliza mecanismos de abstração e decomposição que possibilitam tratar adequadamente a modelagem de sistemas distribuídos e concorrentes. Na maioria das vezes, os sistemas distribuídos são caracterizados por topologias de interconexão dinâmica, ou seja, a topologia destes sistemas pode ser alterada dinamicamente durante seu funcionamento. A abstração oferecida pela orientação a objetos é adequada para a modelagem de tais sistemas, pois permite a criação e destruição dos componentes dinamicamente. Além disso, OO oferece uma modelagem direta das entidades do sistema, conceitos de abstração, hierarquia e o aproveitamento de modelos (reusabilidade).

Embora muitas extensões de OO tenham sido propostas às redes de Petri de alto nível, essas extensões não resolveram apropriadamente o problema da estruturação [Gue98]. A principal razão é que a integração dos conceitos de OO e redes de Petri foi feita de tal maneira que os conceitos de ambos os formalismos foram modificados. Em muitos casos, os conceitos de redes de Petri foram modificados, não permitindo a adaptação dos conhecidos métodos de análise.

Seguindo essa idéia de integração foi desenvolvido no Laboratório de Redes de Petri do Departamento de Ciência da Computação da Universidade Federal da Paraíba, uma linguagem de modelagem de sistemas baseada em redes de Petri e orientação a objetos denominada de RPOO [Gue02], para modelar, investigar e projetar sistemas distribuídos e concorrentes, permitindo a utilização dos mecanismos de estruturação e controle de complexidade oferecidos pela orientação a objetos, e a utilização da base matemática das redes de Petri para a análise e verificação dos sistemas modelados. O formalismo RPOO integra de forma ortogonal esses dois formalismos, adotando uma abordagem complementar, que considera os aspectos de um paradigma que complementam o outro e vice-versa. Seguindo essa aborda-

gem, o sistema modelado pode ser observado a partir de duas visões ortogonais: uma visão de redes de Petri e uma visão orientada a objetos (Figura 1.1). Em cada uma dessas visões, somente os aspectos relacionados ao correspondente paradigma são destacados. Desse modo, a formalização de RPOO preserva as características originais tanto das redes de Petri como da orientação a objetos.

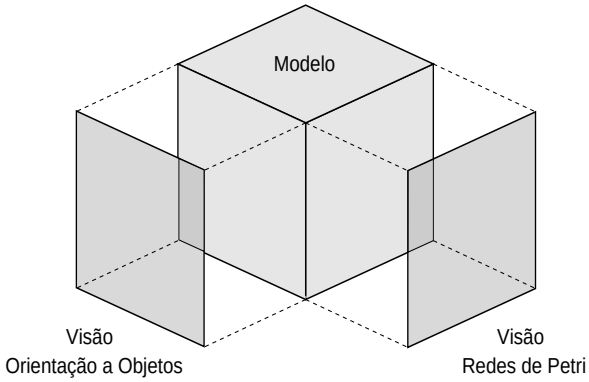


Figura 1.1: Visões Ortogonais dos Aspectos de um Modelo RPOO

Embora a formalização RPOO estivesse concluída no início deste trabalho, não existia nenhuma aplicação prática utilizando a linguagem. Havia apenas um experimento de modelagem em fase inicial sendo elaborado neste período. Assim, percebemos a necessidade de aplicar a notação na modelagem de sistemas reais, já que não havia nenhum modelo que pudesse servir de guia/suporte para a construção de modelos RPOO.

Neste trabalho iremos efetuar um estudo e experimentação da linguagem RPOO, aplicando-a a diversos casos de modelagem. Para realizar este estudo desenvolvemos alguns experimentos, com o objetivo de verificar se a linguagem atende aos propósitos para os quais ela foi projetada. Um estudo comparativo de modelagem também contribui com a experimentação de RPOO. Assim, iremos apresentar um estudo comparativo de modelagem usando o padrão estabelecido *de facto* para estruturação e decomposição, que é HCPN, e a abordagem RPOO. Vale ressaltar, que a comparação de metodologias é uma atividade complicada, pelo fato de cada metodologia ter seu próprio conjunto de definições de conceitos, técnicas e notações. A comparação de uma metodologia é portanto subjetiva a percepção e interpretação da pessoa que executa esta tarefa [HBG93].

A hipótese de trabalho consiste em que RPOO é adequada para modelar sistemas distribuídos, caracterizados por topologias de interconexão dinâmica. Desta forma, os experimen-

tos desenvolvidos neste trabalho pertencem a esta classe de sistemas.

1.2 Objetivos

O objetivo geral deste trabalho é contribuir com o projeto de pesquisa e desenvolvimento da linguagem RPOO. O objetivo específico é efetuar um estudo da linguagem aplicado a um contexto prático, verificando se RPOO é uma boa alternativa para a estruturação e decomposição, em se tratando de sistemas de software distribuídos e concorrentes. Esta verificação será feita através do estudo da linguagem RPOO e da análise experimental dos processos e produtos de modelagem obtidos.

1.3 Metodologia

Para efetuarmos a experimentação de RPOO em um contexto prático, desenvolvemos alguns experimentos de modelagem. Como primeiro experimento escolhemos o Serviço Bouncer – Um Serviço Distribuído para Controle de Licenças de Software[Bez96; BBF97]. O segundo experimento é o sistema *DistBeta*, que é um *framework* para execução de programas distribuídos desenvolvido na linguagem de programação orientada a objeto BETA [JM95; Jen92a]. Para o terceiro experimento escolhemos o protocolo OSPF – *Open Shortest Path First*, que é um protocolo de roteamento especialmente projetado para o ambiente internet, o qual roteia pacotes de um endereço de origem para um endereço de destino, baseando-se apenas no endereço IP. O detalhamento completo do primeiro experimento encontra-se no Capítulo 3 e 4, e os outros dois serão detalhados nos Apêndices B e C.

Durante o desenvolvimento deste trabalho vamos analisar experimentos desenvolvidos por outras equipes de modeladores, para evitar tendências de um único modelador na decomposição e estruturação de sistemas. Dentro do processo de validação de RPOO vamos modelar os experimentos escolhidos usando as abordagens RPOO e HCPN para fazermos um estudo comparativo de modelagem entre os produtos obtidos para ambas as abordagens. No desenvolvimento de cada experimento terei uma atuação diferente. Em alguns momentos serei o modelador, e em outros irei orientar as outras equipes de desenvolvimento. No primeiro experimento participei das duas etapas de modelagem, utilizando a linguagem HCPN

e RPOO. O segundo experimento foi desenvolvido por duas equipes de modeladores diferentes, uma equipe em HCPN e uma em RPOO. Participei da etapa de modelagem usando RPOO, orientando e supervisionando o desenvolvimento dos produtos de modelagem obtidos. No terceiro experimento participei da etapa de desenvolvimento dos produtos de modelagem utilizando a linguagem RPOO. A modelagem usando a linguagem HCPN foi desenvolvida por outro modelador, onde desempenhei o papel de orientador e supervisor.

O processo de experimentação da linguagem RPOO foi efetuado através da modelagem de sistemas usando as abordagens HCPN e RPOO, a partir de uma mesma especificação. Durante o processo de modelagem identificamos as dificuldades e facilidades oferecidas por cada uma das abordagens utilizadas.

Em ambas as etapas de modelagem, utilizamos o Design/CPN [CJK97] como ferramenta de edição e simulação dos modelos. A abordagem RPOO ainda não possui uma ferramenta completa de suporte computacional para análise e verificação de modelos. A validação dos modelos RPOO foi efetuada usando o simulador da ferramenta Design/CPN para simular cada classe isoladamente e a ferramenta SSO [San01] para efetuar a simulação do sistema de objetos.

1.4 Relevância do Trabalho

Neste trabalho realizamos um estudo e experimentação da linguagem de modelagem de sistemas baseada em redes de Petri e Orientação a Objetos, onde verificamos que a linguagem possui uma notação de fácil compreensão. Além disso, os construtos da notação foram suficientes para modelar os sistemas escolhidos. Isto nos dá indícios de que a linguagem RPOO atende às necessidades para as quais foi projetada.

Este trabalho também verificou que RPOO é uma boa alternativa para decompor e estruturar sistemas. O uso de RPOO na modelagem de sistemas distribuídos e concorrentes nos ofereceu maior facilidade para efetuar a comunicação entre os componentes do sistema.

Durante o desenvolvimento dos experimentos tomamos o cuidado de identificar os elementos envolvidos e os problemas encontrados. Dentre os problemas encontrados está a falta de uma ferramenta de software de suporte completa. O desenvolvimento de uma ferramenta de suporte computacional para RPOO facilitará a verificação e análise dos modelos.

Com os experimentos desenvolvidos teremos exemplos que poderão servir de guia/exemplo a outros modeladores na construção de modelos.

É importante observar que um processo de validação de uma linguagem de modelagem não pode ser definitivo apenas com o desenvolvimento de três experimentos. Este trabalho é o ponto de partida para a validação da notação RPOO.

1.5 Estrutura da Dissertação

Esta dissertação está estruturada em quatro capítulos além deste:

Capítulo 2: Embasamento Teórico Neste capítulo apresentamos uma descrição informal através de um exemplo de modelagem simples os conceitos e a utilização das redes de Petri coloridas, redes de Petri coloridas hierárquicas e redes de Petri orientadas a objetos.

Capítulo 3: Experimentos de Modelagem com RPOO Este capítulo é dedicado à apresentação dos experimentos desenvolvidos durante este trabalho utilizando a linguagem RPOO. Inicialmente, apresentamos um detalhamento da modelagem do primeiro experimento, que é o Serviço *Bouncer*. Em seguida, são apresentados os outros experimentos, que são o Sistema *DistBeta* e o protocolo *OSPF* – Open Shortest Path First. Para cada experimento é apresentado uma análise dos resultados obtidos. Os modelos completos do Sistema *DistBeta* e do protocolo *OSPF* encontram-se no Apêndice B e C.

Capítulo 4: Estudo Comparativo Neste capítulo apresentamos um estudo comparativo dos produtos de modelagem obtidos utilizando as abordagens HCPN e RPOO. Inicialmente, apresentamos o detalhamento da modelagem do serviço *Bouncer* usando HCPN e as impressões sobre a solução obtida. Em seguida, é apresentado o estudo comparativo dos modelos obtidos em HCPN e em RPOO, para os três experimentos.

Capítulo 5: Conclusão Finalmente, apresentamos nossas conclusões sobre o uso da linguagem RPOO em um contexto prático, bem como as dificuldades e facilidades enfrentadas durante o processo. Apresentamos também os trabalhos futuros originados da notação RPOO.

Apêndices Nos apêndices são apresentados os produtos de modelagem dos sistemas utilizados para experimentar a notação RPOO na prática. No apêndice A, temos o modelo completo do sistema distribuído para controle de licenças de software – Serviço *Bouncer*. No apêndice B é apresentado um detalhamento da modelagem do sistema *DistBeta* e finalmente no apêndice C é detalhado a modelagem do protocolo *OSPF*.

Capítulo 2

Embasamento Teórico

O conceito de redes de Petri (*PN - Petri Nets*) surgiu em 1962, na tese de doutorado de *Carl Adam Petri*, intitulada *Kommunikation mit Automaten*, na Alemanha. As redes de Petri são uma ferramenta matemática para a modelagem e análise de sistemas, especialmente os que possuem características concorrentes, distribuídas, paralelas, assíncronas e/ou estocásticas [Mur89]. As redes de Petri possuem uma notação gráfica bastante amigável e seus modelos são ditos executáveis, por permitirem simulações.

Uma rede de Petri é composta de uma estrutura de rede, inscrições associadas a essa estrutura e uma marcação. A estrutura de rede é um grafo bipartido direcionado, com dois tipos de nós: lugares, representados graficamente por círculos, e transições, representadas graficamente por retângulos. A marcação de uma rede de Petri é definida pela presença de zero ou mais fichas em um lugar. A *marcação inicial* de uma rede representa o estado inicial do sistema. A representação dos estados do sistema é distribuída nos lugares da rede, enquanto as ações são representadas pelas transições.

A Figura 2.1 representa uma rede de Petri composta por cinco lugares: *P1*, *P2*, *P3*, *P4* e *P5*, e quatro transições: *T1*, *T2*, *T3* e *T4*. Esta rede modela uma situação em que dois tipos de processos: *A* e *B*, utilizam recursos do tipo *R*. A marcação inicial desta rede indica que temos uma ficha nos lugares *P1* e *P2* e duas fichas no lugar *P4*, representando um processo *A*, um processo *B*, e dois recursos *R*, respectivamente. O arco que liga o lugar *P4* com a transição *T1* possui peso *2*, significando que o processo *A* precisa de *dois* recursos *R* para executar. Os arcos sem inscrição explícita possuem peso 1. Logo, o processo *B* necessita apenas de *um* recurso R para executar.

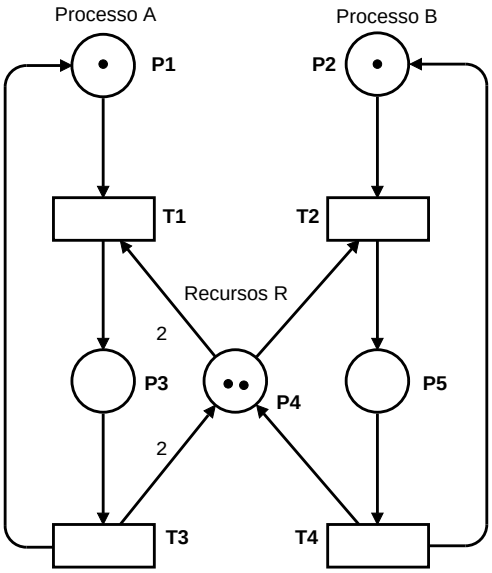


Figura 2.1: Exemplo de uma rede de Petri

As transições de uma rede possuem um certo número de lugares de entrada, que são os lugares de origem de onde partem os arcos que atingem a transição; e de lugares de saída, que são os lugares para onde partem os arcos da transição. Assim, a transição *T1* possui o lugar *P1* como lugar de entrada e o lugar *P3* como lugar de saída.

O comportamento de um sistema modelado em redes de Petri pode ser descrito em termos dos estados do sistema e de suas alterações. Para simular o comportamento dinâmico de um sistema, um estado ou marcação em uma rede de Petri é alterado de acordo com uma regra de disparo. Uma regra de disparo, portanto, determina quais são as condições para uma transição estar habilitada a disparar/ocorrer e quais as consequências do seu disparo. A regra de disparo de uma transição pode ser dividida em três partes:

1. Uma transição está habilitada a disparar, se cada lugar de entrada contém pelo menos o número de fichas indicado pelo peso do arco do respectivo lugar de entrada.

2. Uma transição habilitada pode ou não disparar.

3. O disparo de uma transição habilitada remove o número de fichas indicado pelo peso do arco de cada lugar de entrada, e adiciona no lugar de saída o número de fichas indicado pelo peso do arco do referido lugar.

De acordo com a regra de disparo as transições *T1* e *T2* estão habilitadas a ocorrer. O

disparo da transição *T1* indica que o processo *A* consumiu dois recursos *R*. O estado do sistema após o disparo desta transição alterou para uma ficha no lugar *P3* e uma ficha no lugar *P2* – Figura 2.2. Neste novo estado do sistema, apenas a transição *T3* está habilitada a disparar.

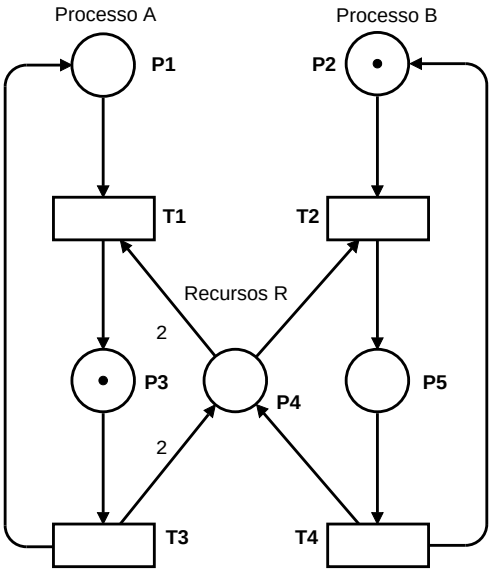


Figura 2.2: Novo Estado após o Disparo da Transição *T1*

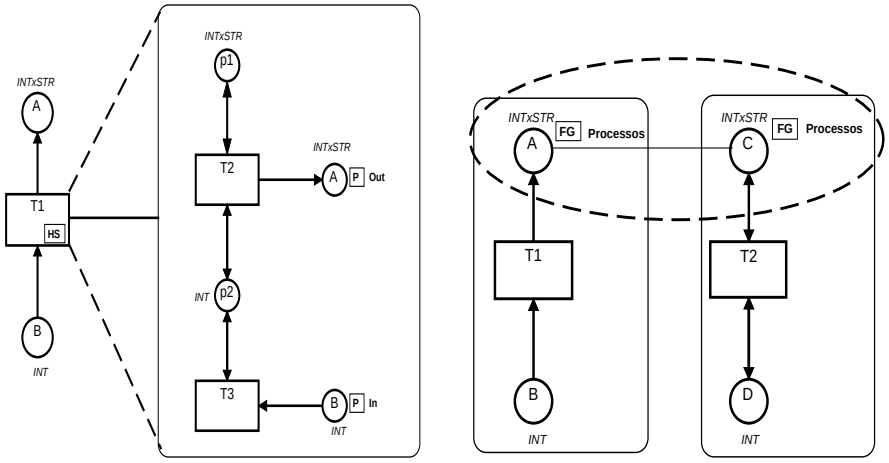
Na rede de Petri apresentada na Figura 2.1, as inscrições associadas aos arcos da estrutura da rede definem apenas os pesos desses arcos. As fichas associadas aos lugares representam dados binários. Entretanto, há diversas limitações ao utilizá-las para modelar sistemas complexos. Uma delas é a necessidade de replicação da estrutura da rede para modelar processos idênticos, o que tende a aumentar o tamanho dos modelos. Observe na figura que a estrutura da rede que modela o processo *A* é igual a do processo *B*. Essa replicação ocorre porque suas fichas não carregam tipos de dados, assim não podemos diferenciá-las. Para suprir tais limitações e facilitar a modelagem de sistemas maiores, extensões foram propostas as redes de Petri. Surgindo assim, a classe das redes de Petri de alto nível.

As redes de Petri de alto nível caracterizam-se por permitirem associar tipos de dados às fichas. As fichas podem carregar informações complexas, que são manipuladas conforme as inscrições dos arcos. O fato das fichas expressarem informações complexas aumenta o poder de descrição dessas redes e, consequentemente, podemos construir modelos mais compactos. A passagem de redes de Petri de baixo nível para redes de alto nível pode ser comparada à passagem da linguagem de máquina para linguagens de programação de alto nível [Jen92c].

Devido à existência de uma ferramenta de software de suporte (Design/CPN [CJK97]), as redes de Petri coloridas (CP-Net) são as redes mais conhecidas dentre as redes de alto nível.

Mesmo com fichas mais elaboradas para construir modelos grandes usando redes de Petri de alto nível são necessários mecanismos de estruturação e conceitos de abstração, que permitam trabalhar com uma parte selecionada do modelo sem nos deixar influenciar pelos detalhes de baixo nível das demais partes.

Uma solução para este problema baseia-se na utilização de hierarquia para estruturar os modelos. Em termos de CP-Net, esta abordagem resultou na linguagem denominada redes de Petri coloridas hierárquicas (HCPN) [Jen92c]. HCPN permite ao modelador a construção de modelos grandes combinando um número de pequenas redes de Petri coloridas em uma grande rede. A modelagem de sistemas usando HCPN pode ser desenvolvida usando uma abordagem top-down ou bottom-up. A HCPN possui dois mecanismos de estruturação: transições de substituição e lugares de fusão. As transições de substituição representam uma abstração de uma determinada parte do modelo (transição de substituição *T1* da Figura 2.3(a)). Lugares de fusão permitem ao usuário especificar um conjunto de lugares que são idênticos (lugares de fusão *A* e *C* da Figura 2.3(b). Quando uma ficha é adicionada/removida de um lugar de fusão, uma ficha idêntica será adicionada/removida em todos os outros lugares do modelo.



(a) Exemplo de uma Transição de Substituição

(b) Exemplo de um Lugar de Fusão

Figura 2.3: Mecanismos de Estruturação das HCPN

Uma outra abordagem proposta para resolver o problema da estruturação é a integração dos conceitos de orientação objeto (OO) na teoria de redes de Petri [Lak95; BB88; ELR90; Gue97]. Dentro dessa abordagem encontramos as redes de Petri Orientadas a Objetos (RPOO) [Gue02]. RPOO utiliza os conceitos da orientação objeto para estruturar os sistemas e as CP-Net para detalhar o comportamento dos componentes. A união das abordagens orientação a objetos e redes de Petri permite a utilização dos mecanismos de estruturação e controle de complexidade oferecidos pela orientação a objetos, e a utilização da base matemática das redes de Petri coloridas para tratar de sistemas complexos. Devido ao fato de RPOO utilizar CP-Net, começaremos a próxima seção a partir da descrição destas redes, objetivando facilitar a compreensão de RPOO.

Este capítulo está organizado como segue. Na Seção 2.1 será introduzido as redes de Petri coloridas. Na Seção 2.2 apresentamos as redes de Petri coloridas hierárquicas. Finalmente, na Seção 2.3 será apresentado as redes de Petri Orientadas a Objetos.

2.1 Redes de Petri Coloridas – CP-Net

Apesar das redes de Petri serem uma ferramenta formal, iremos apresentá-las através de um exemplo informal. Vamos utilizar o exemplo de um protocolo simples, como definido em [Jen]. Esse protocolo considera duas máquinas que trocam mensagens através de um canal de comunicação não confiável. O protocolo garante que os dados enviados são entregues com sucesso, na ordem que foram enviados e sem duplicação. Uma das máquinas (emissor) envia pacotes de dados e utiliza uma estratégia de retransmissão periódica para garantir o envio dos dados com sucesso. O emissor seleciona um novo pacote de dados quando recebe o reconhecimento da recepção do último pacote enviado. A máquina que recebe os pacotes (receptor) deve guardar apenas uma cópia de cada pacote. Ao receber um pacote de dados com sucesso, o receptor envia o reconhecimento, solicitando o próximo pacote de dados.

Em uma rede de Petri colorida cada ficha tem associada a ela um valor de dado chamado cor da ficha. Associado a cada lugar há um tipo de dados chamado de *colour set*—conjunto de cores, que determina os possíveis valores das fichas ali contidas. A declaração dos conjuntos de cores associados aos lugares, variáveis e funções utilizadas nas redes dos modelos são feitas no *nó de declaração*. Em CP-Net, a linguagem de programação utilizada nas

inscrições e nas declarações é denominada CPN-ML, que é uma extensão da linguagem de programação *Standard ML* [D.U94].

Na Figura 2.4 encontra-se o *nó de declarações* usado no modelo, que contém a declaração dos conjuntos de cores (*color INT, DATA, INTxDATA, Ten0* e *Ten1*), variáveis (*n, k, p, str, r* e *s*), constantes (*stop*) e funções (*fun Ok*) utilizadas no modelo. Os conjuntos de cores estão associados aos lugares da rede, a função e as variáveis são utilizadas nas inscrições de arco do modelo. A *fun OK* é responsável por determinar se um pacote de dados será enviado com sucesso, ou se o mesmo será perdido. Esta função indica que o pacote será perdido quando o valor da variável *r* for maior que o valor da variável *s*. Como *r* pode assumir os valores de 1 a 10 e *s* somente recebe o valor 8, existe vinte por cento de chances do pacote de dados ser perdido.

```
color INT = int;
color DATA = string;
color INTxDATA = product INT*DATA;
var n,k:INT;
var p,str:DATA;
val stop = "########";

color Ten0 = int with 0..10;
color Ten1 = int with 0..10;
var s:Ten0; var r:Ten1;
fun Ok(s:Ten0,r:Ten1)=(r<=s);
```

Figura 2.4: Nó de Declarações

Na Figura 2.5 é apresentada uma CP-Net para a modelagem do protocolo, o qual consiste de três partes: *Emissor*, *Rede* e *Receptor*. As expressões de inicializações associadas aos lugares *Send*, *NextSend*, *Received* e *NextRec*, obedecem ao conjunto de cores desses lugares e estabelecem a presença das fichas, como podemos observar nas expressões próximas a eles. A parte *Emissor* consiste de dois lugares *Send* e *NextSend* e duas transições *Send Packet* e *Receive Acknow*. O conjunto de cores do lugar *Send* é composto por um par, onde o primeiro elemento é um inteiro e o segundo uma *string*, representando respectivamente o número e o conteúdo do pacote de dados. A marcação inicial deste lugar contém a sequência de pacotes que serão transmitidos. Logo, o primeiro pacote a ser transmitido será o pacote *1* contendo o seguinte dado: *"Estudo e"*. O lugar *NextSend* contém o número do próximo pacote a ser enviado. Inicialmente o número do pacote é 1, que será atualizado cada vez que um

reconhecimento de um pacote for recebido.

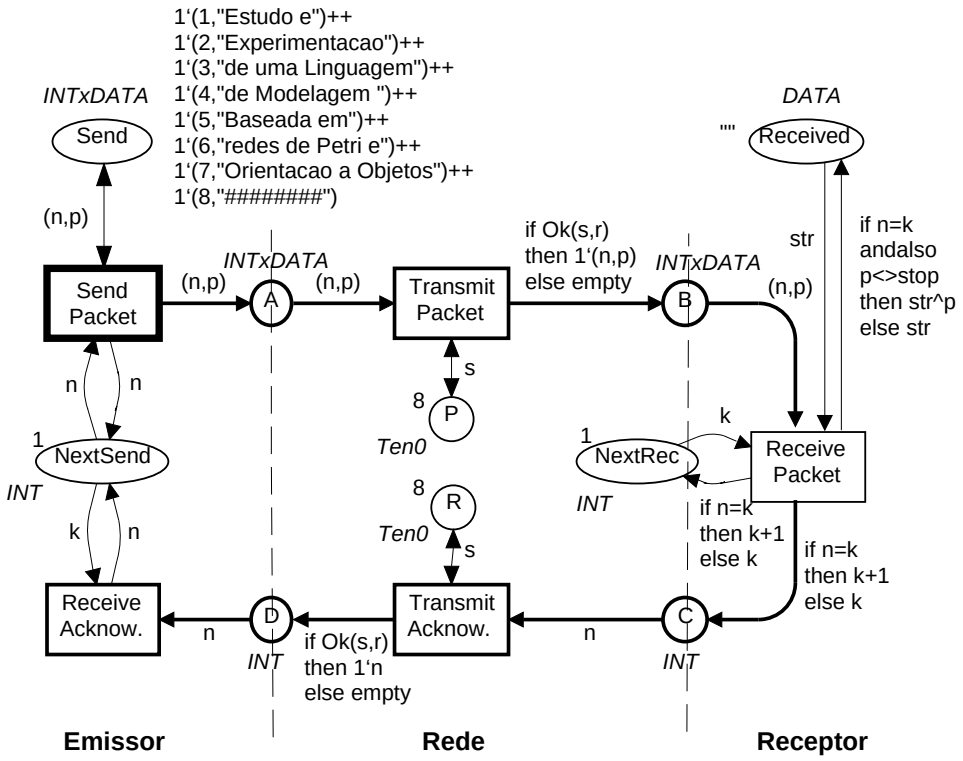


Figura 2.5: Exemplo de uma rede de Petri colorida

A parte *Rede* é composta de dois lugares: *P* e *R* e duas transições *Transmit Packet* e *Transmit Acknow*. Essas transições modelam o envio e o reconhecimento de um pacote de dados. As fichas armazenadas nos lugares *P* e *R* contém o valor da variável *s*, que será utilizada na função *Ok* para determinar se o pacote será enviado com sucesso ou se será perdido.

A parte *Receptor* é composta de dois lugares : *Next Rec* e *Received*, e uma transição *Receive Packet*. O conjunto de cores do lugar *Next Rec* é composto por um inteiro, representando o número do próximo pacote a ser recebido da parte *Emissor*. A marcação inicial deste lugar contém o número do primeiro pacote a ser recebido, que é *1*. Os pacotes recebidos do emissor serão armazenados no lugar *Received*. Inicialmente, a ficha deste lugar é uma *string* vazia.

Os lugares *A* e *D* servem de interface entre as partes *Emissor* e *Rede* e os lugares *B* e *C* servem de interface entre as partes *Rede* e *Receptor*.

Para entender a regra de disparo de uma CP-Net, é necessário compreender os conceitos

de variável de transição, ligação (*binding*), e elemento de ligação. As variáveis de uma transição são todas as variáveis encontradas nas inscrições de arcos ligados a esta transição. Por exemplo, a transição *Send Packet* tem duas variáveis: *n* e *p*. Uma ligação é a associação das variáveis de uma transição a valores dos seus conjuntos de cores. Por exemplo, uma das possíveis ligações para as variáveis de transição *(n,p)* é : (<*n=1*>,<*p=“Estudo e”*>). Um elemento de ligação é composto por uma transição e uma ligação para esta transição. Para que um elemento de ligação esteja habilitado é necessário que exista, nos lugares de entrada da respectiva transição, fichas correspondentes às avaliações das expressões de arco, considerando a ligação das variáveis de transição. Na Figura 2.5, o elemento de ligação, composto pela transição *Send Packet* e ligação (<*n=1*>,<*p=“Estudo e”*>) está habilitado a disparar para a marcação inicial do modelo. Um elemento de ligação habilitado pode ou não disparar. O disparo de um elemento de ligação remove fichas corretas dos lugares de entrada e adiciona aos lugares de saída. As fichas adicionadas/removidas são determinadas pelas expressões de arco da respectiva transição.

Quando a transição *Send Packet* disparar, um pacote de dados será enviado para a rede, através da adição de uma ficha com uma cópia do pacote no lugar *A* (inscrição de arco *(n,p)*). A ficha com o referido pacote não é removida do lugar *Send* porque o pacote pode ser perdido, portanto uma retransmissão do mesmo será necessária. A transição *Receive Acknow* somente irá disparar quando o emissor receber a confirmação do receptor, atualizando a ficha do lugar *NextSend* com o número do próximo pacote a ser enviado.

Quando a transição *Transmit Packet* disparar um pacote será transmitido da parte Emissor da rede para a parte Receptor, removendo a ficha do lugar *A* e adicionando no lugar *B*. As inscrições de arco desta transição determinam se o pacote será enviado com sucesso ou se será perdido (a perda de um pacote é determinada pela *função Ok*). A transição *Transmit Acknow* transmite a confirmação do recebimento de um pacote da parte Receptor da rede para a parte Emissor, removendo a ficha do lugar *C* e adicionando no lugar *D*.

Ao disparar a transição *Receive Packet* será verificado o número do pacote recebido, caso seja o número do pacote esperado, a ficha do lugar *NextRec* será atualizada com o número do próximo pacote a ser recebido (inscrição de arco *if n=k then k+1 else k*) e os dados do pacote serão concatenados com os dados da ficha do lugar *Received*. Após o recebimento de um pacote, uma ficha será depositada no lugar *C* informando o número do próximo pacote

que a parte *Receptor* deseja receber da parte *Emissor*. Quando o número do pacote recebido não for o número esperado, o pacote será descartado. No final da transmissão dos pacotes o lugar *Received* contém o seguinte texto: *Estudo e Experimentacao de uma Linguagem de Modelagem Baseada em Redes de Petri e Orientacao Objetos*, como pode ser observado na Figura 2.6.

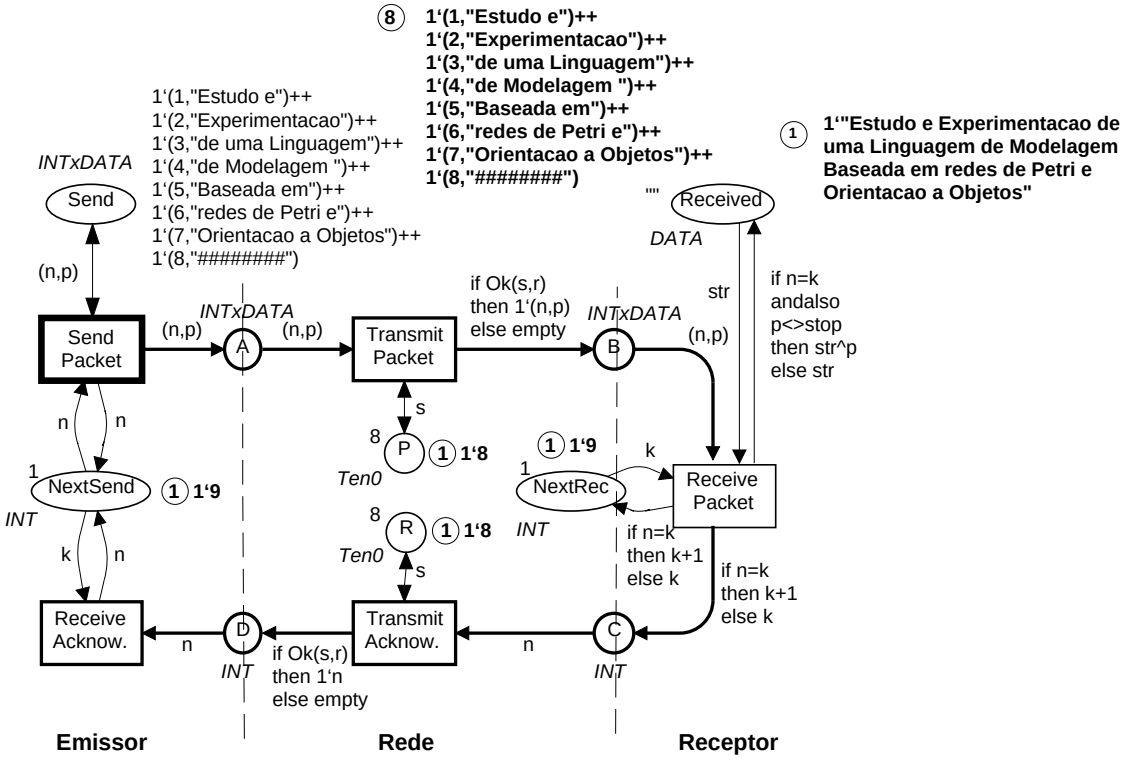


Figura 2.6: CP-Net após a Transmissão de Todos os Pacotes

Uma explicação mais detalhada sobre as redes de Petri coloridas pode ser encontrada em [Jen92c].

2.2 Redes de Petri Coloridas Hierárquicas – HCPN

Vejamos agora como o exemplo do protocolo foi construído utilizando a linguagem HCPN para modelar e estruturar o modelo. Cada parte do protocolo foi modelada em uma rede separada.

Na Figura 2.7 é apresentada a página de hierarquia do modelo HCPN. Os retângulos com os vértices arredondados representam as redes (páginas) do modelo, que possui um total de

05 páginas. As páginas são ligadas por arcos direcionados que indicam a relação de hierarquia. Como podemos observar na figura as páginas *Emissor, Rede* e *Receptor* estão um nível abaixo da página *ProtocoloSimples*. Estas páginas são consideradas subpáginas da página *ProtocoloSimples*, ou seja, são as páginas que contém o detalhamento das atividades modeladas pelas transições de substituição encontradas na página *ProtocoloSimples*. As inscrições de arco *Emissor, Rede* e *Receptor* correspondem ao nome da transição de substituição que a subpágina representa. A página *nodedeclaracao* contém o *nó de declarações* utilizado no modelo, que é o mesmo do modelo usando CP-Net.

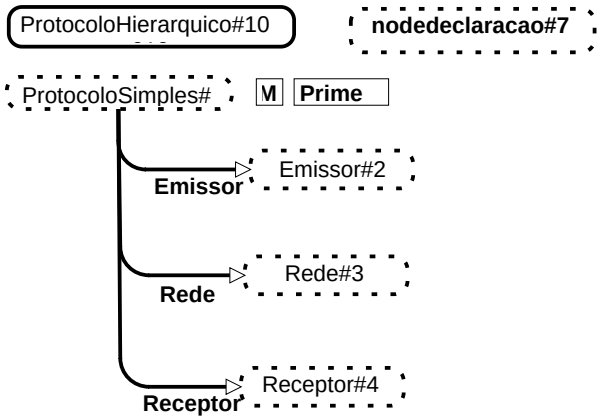


Figura 2.7: Página de Hierarquia

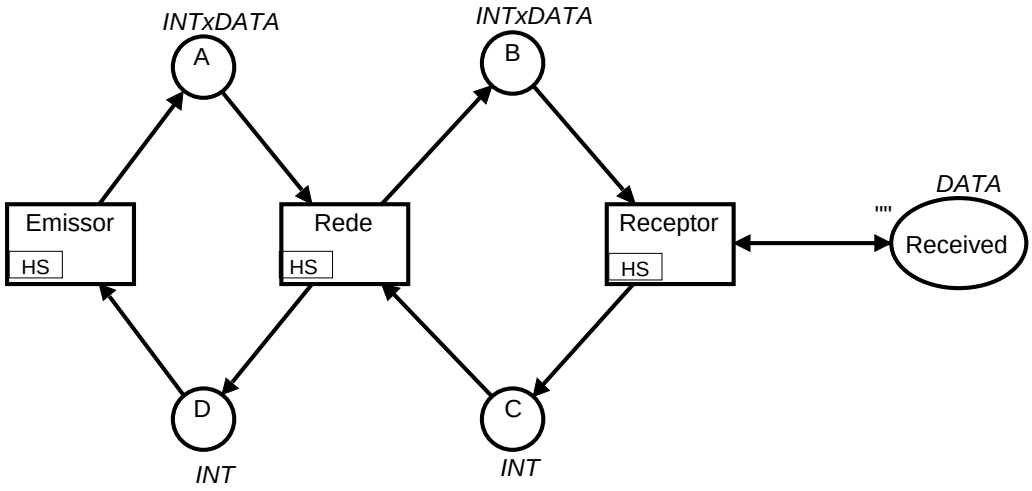


Figura 2.8: Página ProtocoloSimples

Na Figura 2.8 é apresentada a página *ProtocoloSimples*. Nesta rede podemos identificar três elementos do protocolo que são modelados pelas transições de substituição: *Emissor*,

Rede e *Receptor*. Estas transições representam uma abstração das três partes do protocolo. Os lugares circunvizinhos de uma transição de substituição são chamados de *sockets*. Logo, os lugares *A* e *D*, *B* e *C*, e *Received* são *sockets* das respectivas transições. Os *sockets* descrevem a interface de uma página com as subpáginas que detalham as transições de substituição. Cada *socket* tem uma porta (lugar) equivalente. O lugar *socket* tem sempre a mesma marcação de sua correspondente porta.

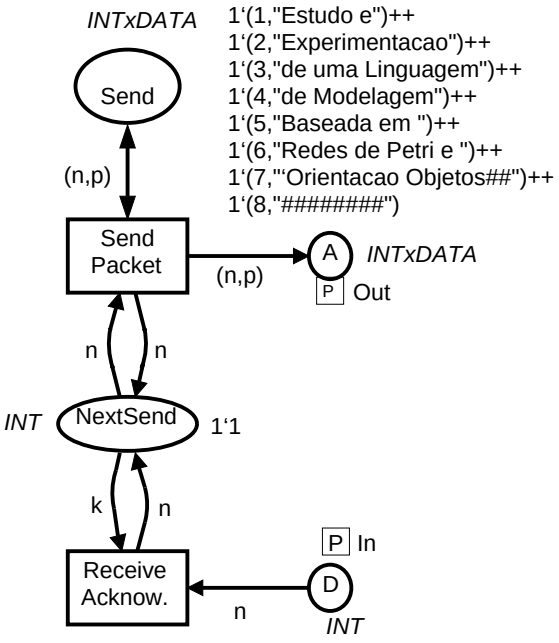


Figura 2.9: Parte Emissor

Na Figura 2.9 é apresentada a parte do modelo que detalha o *Emissor*. A porta que corresponde ao lugar *A* é chamada de porta de saída (*P Out*). Logo, o pacote saíra da rede que modela a parte *Emissor* para a rede que modela a parte *Rede* através do lugar *A*. A porta que corresponde ao lugar *D* é uma porta de entrada (*P Int*). Ou seja, a confirmação do recebimento dos pacotes será recebida da rede por este lugar. Note que todo o funcionamento da parte *Emissor* está modelado nesta rede. O comportamento da parte *Emissor* usando HCPN é idêntico ao modelo utilizando CP-Net.

Na figura 2.10 é apresentada uma CP-Net modelando a parte *Rede*. Como podemos observar a estrutura desta rede é igual a parte Rede apresentada na Figura 2.5. Os lugares *A* e *C* são portas de entrada. Estes lugares servem de interface de entrada para a parte *Emissor* e *Receptor*. A rede enviará os pacotes para o emissor/receptor através dos lugares *B* e *D*, que

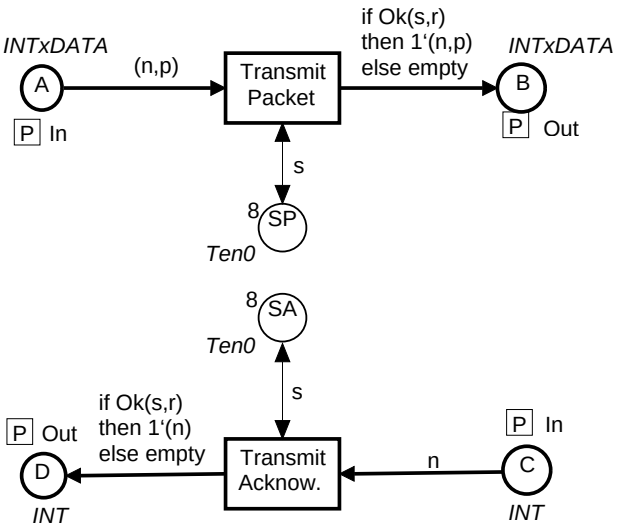


Figura 2.10: Parte Rede

são portas de saída.

Na figura 2.11 é apresentada uma CP-Net modelando a parte *Receptor*. A estrutura desta rede também é igual a parte Receptor modelada em CP-Net. Observe que todas as redes do modelo HCPN correspondem a uma parte da rede original do modelo usando CP-Net—Figura 2.5. No modelo HCPN cada componente foi modelado separadamente em uma única rede.

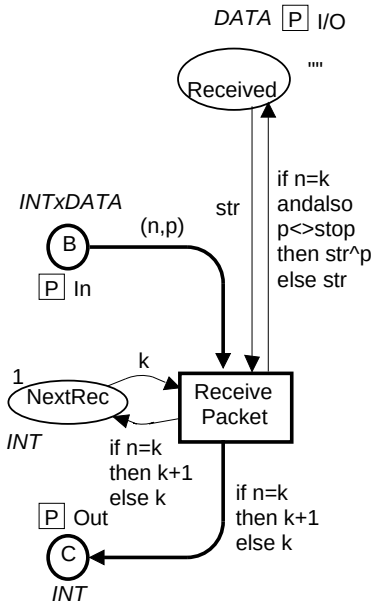


Figura 2.11: Parte Receptor

2.3 Redes de Petri Orientadas a Objetos – RPOO

A linguagem RPOO utiliza as construções da orientação objeto para estruturar os modelos. Logo, a construção chave da notação é a classe, e cada classe descreve uma entidade do sistema. Os objetos, que são as instâncias das classes, modelam entidades autônomas e concorrentes. Assim, um modelo RPOO consiste de um diagrama de classes, representando as entidades do sistema e suas associações; um diagrama de fluxo de mensagens, que contém todas as mensagens trocadas entre os objetos do sistema; e um diagrama de configuração inicial contendo as instâncias das classes existentes na iniciação do modelo. O detalhamento de uma classe em RPOO consiste em uma rede de Petri colorida, que é o corpo da classe, e inscrições de interação associadas às transições do corpo da classe, que viabilizam a interação entre os objetos do sistema. O corpo da classe define o comportamento dos objetos da referida classe. As ações são representadas por transições, e a representação dos estados é distribuída nos lugares da rede.

RPOO utiliza sistemas de objetos, que são coleções de objetos capazes de interagir para atender a um propósito, para modelar sistemas de software concorrentes e distribuídos [Gue02]. O sistema de objetos propõe um conjunto de regras que define como um sistema é representado a partir da comunicação entre as instâncias que estão executando.

Na Figura 2.12, é apresentado uma visão abstrata de um modelo RPOO. Cada círculo dentro do *Sistema de Objetos* representa a instância de uma classe. Logo, temos quatro instâncias: *objeto1*, *objeto2*, *objeto3* e *objeto4*. As ligações entre os objetos indicam a possibilidade de envio/recepção de mensagens entre eles. Como os objetos são entidades autônomas, o envio de uma mensagem de um objeto para outro não implica no consumo da mensagem. Assim, mensagens podem ficar pendentes e por isso são elementos considerados na formalização do sistema de objetos. O conjunto de objetos, ligações entre os objetos e mensagens pendentes é chamado de *estrutura* do sistema de objetos. As regras do sistema de objetos indicam qual o efeito do disparo da ação de um objeto sobre uma estrutura, ou seja, as regras definem como calcular a nova estrutura do sistema de objetos a partir do disparo de uma ação.

As inscrições de interação representam os tipos de ações que servem de entrada para o sistema de objetos e que podem alterar a estrutura do sistema modelado. As ações em

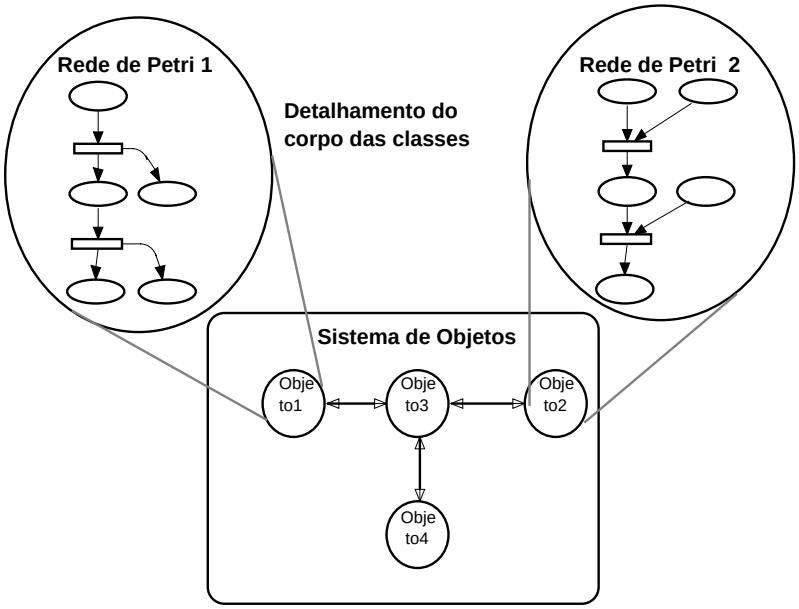


Figura 2.12: Visão Abstrata de um Modelo RPOO

RPOO podem ocorrer em conjunto, ou seja, mais de uma ação pode ser executada de forma atômica por objetos diferentes. Esse conjunto de ações é conhecido como *Evento*. As regras do sistema de objetos definem também qual o efeito do disparo de um evento sobre uma estrutura, a partir da composição do efeito das ações. Existem sete tipos de ações definidas em RPOO, mais especificamente na formalização do sistema de objetos:

1. **Ação interna** – São ações ocorridas dentro de um objeto e que não modificam a estrutura do sistema de objetos. No corpo das classes, as transições sem inscrições de interação são interpretadas como uma ação interna.

2. **Ação de instanciação de objetos** – É uma ação na qual um objeto cria uma instância de outro. Quando um objeto é criado, ele passa a fazer parte da estrutura do sistema de objetos e uma ligação entre o objeto criador e o objeto criado também é inserida na estrutura.

3. **Ação de saída assíncrona de dados** – Uma ação em que um objeto envia uma mensagem para outro e continua seu processamento. As regras indicam que a mensagem só pode ser enviada se o objeto agente da ação possui uma ligação com o objeto destino. O efeito dessa ação é a criação de uma mensagem pendente na estrutura.

4. **Ação de saída síncrona de dados** – Ação em que um objeto envia uma mensagem

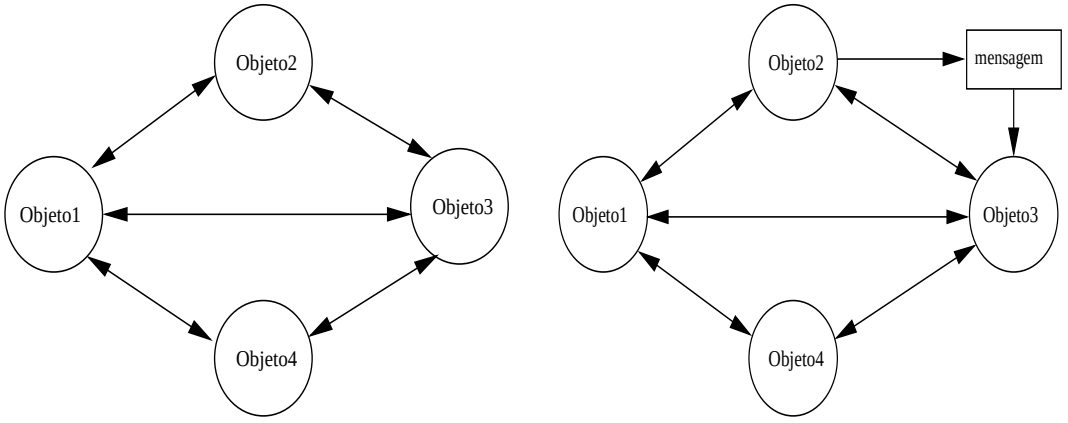
para outro e fica em estado de espera até que o objeto destino consuma a mensagem. O efeito dessa ação é a criação e consumo de uma mensagem na estrutura.

5. **Ação de entrada de dados** – Uma ação em que um objeto consome uma mensagem pendente. A mensagem consumida é excluída da estrutura do sistema de objetos.

6. **Ação de desligamento** – É uma ação em que um objeto se desliga de outro objeto.

7. **Ação de auto-destruição** – Ação em que um objeto se destrói. As regras definem que todas as ligações em que o objeto origem é o agente da ação sejam excluídas da estrutura.

Para exemplificar o efeito do disparo de uma ação sobre uma estrutura do sistema de objetos, vamos considerar a estrutura inicial apresentada na Figura 2.13(a). O disparo de uma ação de saída assíncrona de dados sobre essa estrutura, resulta em uma nova estrutura, como podemos observar na Figura 2.13(b). Esta nova estrutura contém uma mensagem pendente do *Objeto2* para o *Objeto3*.



(a) Estrutura Inicial

(b) Nova Estrutura Depois da Aplicação de uma Ação

Figura 2.13: Estrutura do Sistema de Objetos

A descrição formal das regras de disparo das ações e dos eventos sobre uma estrutura, no sistema de objetos, pode ser encontrada em [Gue02].

2.3.1 Diagrama de Classes

A modelagem em RPOO é feita através do conceito de classes e dos mecanismos para relacioná-las: associações, agregações e especializações.

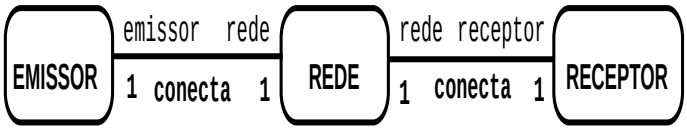


Figura 2.14: Diagrama de Classes

Na Figura 2.14 é apresentado um diagrama de classes RPOO para o exemplo do protocolo. O diagrama de classes representa uma visão abstrata dos objetos do sistema e seus relacionamentos. Os nós do diagrama representam as classes e os arcos representam as associações. As inscrições nas extremidades das associações declaram a cardinalidade e os seletores. Logo, uma rede conecta um emissor e um receptor. Os seletores *emissor, receptor* e *rede* podem ser utilizados no detalhamento das classes para efetuar a troca de mensagens entre os objetos.

2.3.2 Diagrama de Fluxo de Mensagens

Na Figura 2.15 é apresentado o diagrama de fluxo de mensagens para o modelo. Este diagrama pode auxiliar o modelador no detalhamento do corpo das classes. Como pode ser observado nas inscrições de arco da figura, através do diagrama podemos identificar todas as mensagens que podem ser trocadas entre os objetos do modelo. Como por exemplo entre as classes Emissor e Rede, podemos identificar que o Emissor envia a mensagem *Send_PackData* para a Rede. A rede, por sua vez, enviará como resposta a mensagem *ReceiveAck_PackData*.

O diagrama de fluxo de mensagens pode ser gerado a partir do diagrama de classes e do detalhamento dos corpos das classes. Embora a sua modelagem seja opcional, ele foi

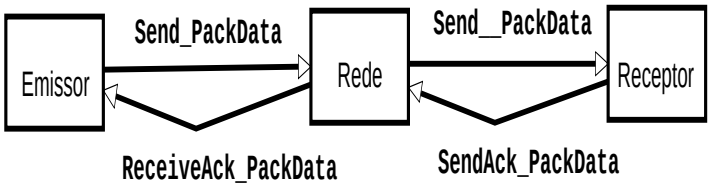


Figura 2.15: Diagrama de Fluxo de Mensagens

descrito antes do detalhamento dos corpos das classes porque ele pode ajudar o modelador a manter consistente os produtos de modelagem.

2.3.3 Diagrama de Configuração Inicial

O diagrama de classes descreve a natureza dos objetos que podem existir no sistema, mas não indica o número de objetos que compõem o sistema. A configuração é feita através do diagrama de configuração inicial. Na Figura 2.16 é apresentado o diagrama de configuração inicial para o nosso modelo. Os nós representam os objetos e os arcos representam as ligações entre os objetos. Para este exemplo temos uma instância da classe *EMISSOR* com o identificador emissor, uma instância da classe *RECEPTOR* com o identificador receptor e uma instância da classe *REDE* com o identificador rede.

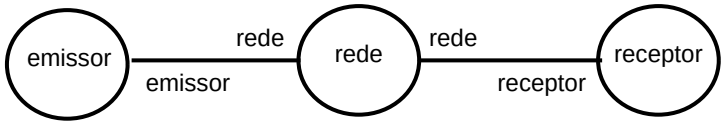


Figura 2.16: Diagrama de Configuração Inicial

2.3.4 Detalhamento das Classes RPOO

No detalhamento do corpo das classes, os seletores são considerados declarações de variáveis. Todas as declarações dos conjuntos de cores, variáveis e funções em RPOO são feitas em nós de declaração – de forma semelhante à usada em redes de Petri coloridas.

Na Figura 2.17 é apresentado o corpo da classe *EMISSOR*. O corpo da classe consiste em uma rede com dois lugares (*Send* e *Wait Ack*) e três transições (*Send Packet*, *Receive Acknow.* e *Next Send*). A ocorrência da transição *Send Packet* indica que o emissor está

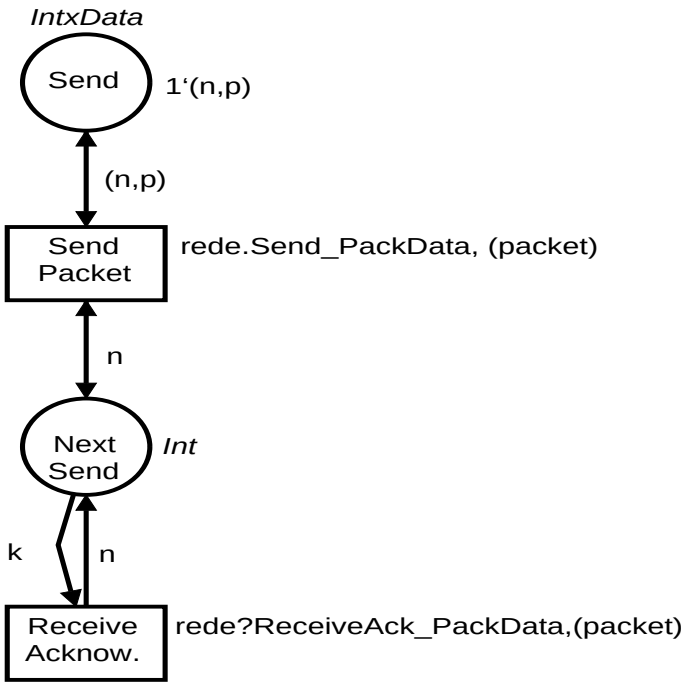


Figura 2.17: Classe EMISSOR

enviando a mensagem *rede.Send_PacketData,(packet)* com um pacote de dados para a rede de maneira assíncrona. A ocorrência da transição *Receive Acknow.* indica que o emissor recebeu a mensagem *rede?ReceiveAck_PackData,(packet)* da rede com o reconhecimento do pacote enviado.

Na Figura 2.18 é apresentado o corpo da classe *REDE*. O corpo da classe consiste em uma rede com dois lugares *free* e *busy* e quatro transições *Receive Receptor, Receive Emissor, Send Emissor* e *Send Receptor*. As transições *Receive Receptor* e *Receive Emissor* têm as seguintes inscrições de interação de entrada *receptor?SendAck_PacketData,(packet)* e *emissor?Send_PacketData,(packet)*. A ocorrência destas transições indicam que a rede recebeu uma mensagem com um pacote de dados do emissor ou uma mensagem com um reconhecimento de um pacote do receptor, respectivamente. Quando a rede recebe uma mensagem do emissor/receptor seu estado mudará para *busy*.

A partir do estado *busy*, podem ocorrer as transições *Send Emissor* e *Send Receptor*, que possuem as inscrições de interação de saída síncrona *emissor!ReceiveAck_PacketData,(packet)* e *receptor!Send_PackData,(packet)*. A ocorrência destas transições indica que a rede enviou uma mensagem para o objeto emissor ou para

o objeto receptor. As guardas[1] *str=receptor* e *str=emissor* associadas a essas transições, garantem que as mensagens serão entregues ao destinatário correto. Ou seja, se a mensagem chegou do emissor, ela será entregue ao receptor. Caso tenha chegado do receptor, a mesma será entregue ao emissor.

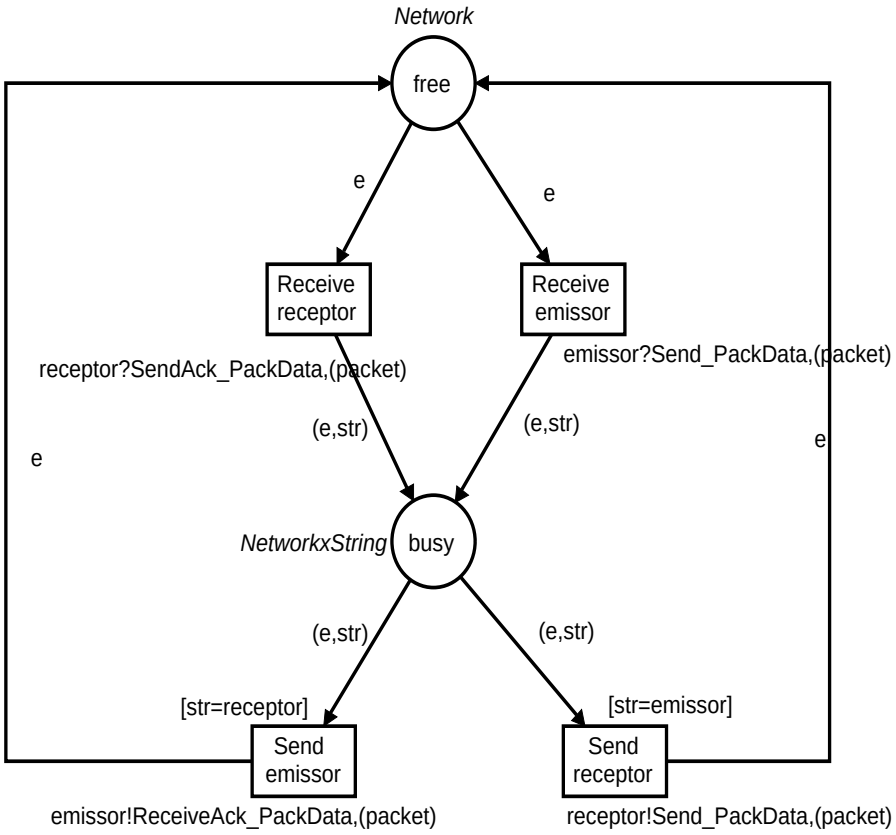


Figura 2.18: Classe REDE

Na Figura 2.19 é apresentado o corpo da classe *RECEPTOR*. O corpo da classe consiste em uma rede com dois lugares *Received* e *Send Acknow.* e duas transições *Receive Packet* e *Send Acknow*. A ocorrência da transição *Receive Packet* indica que o receptor recebeu a mensagem *rede?Receive_PacketData,(packet)* com um pacote de dados da rede. A transição *Send Acknow* modela a situação em que o receptor envia a mensagem *rede.SendAck_PackData,(packet)* de maneira assíncrona para a rede confirmando o recebimento do pacote de dados.

Devido à simplicidade do exemplo, nem todas as inscrições de interação disponíveis da linguagem RPOO foram utilizadas. No capítulo seguinte, apresentamos alguns experimentos

[1]Guardas são expressões booleanas que restringem a ocorrência das transições.

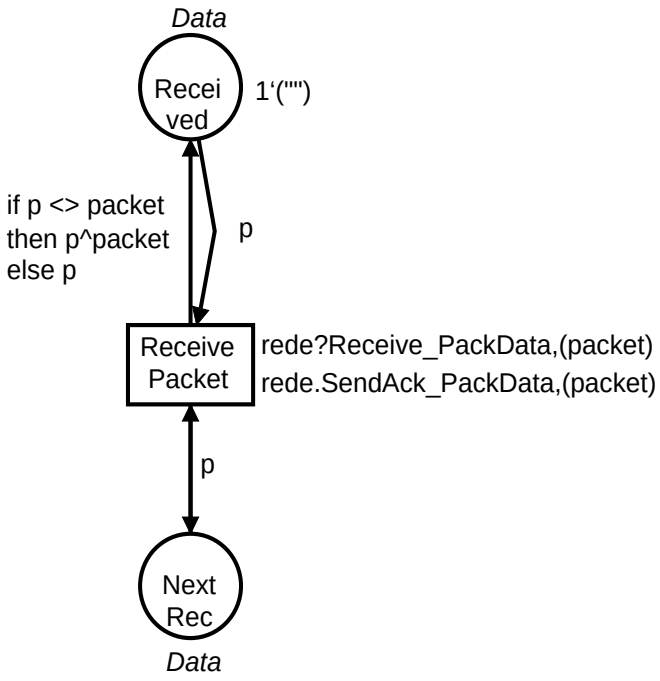


Figura 2.19: Classe RECEPTOR

mais complexos, onde todas as inscrições de interação são usadas.

Capítulo 3

Experimentos de Modelagem com RPOO

Neste capítulo apresentamos os experimentos de modelagem desenvolvidos durante este trabalho utilizando a linguagem RPOO. Estes experimentos foram desenvolvidos com o propósito de aplicar a notação RPOO, verificando como a linguagem se comporta na prática para modelar sistemas de software.

A linguagem RPOO foi desenvolvida para modelar, analisar e verificar sistemas distribuídos e concorrentes, caracterizados por topologias de interconexão dinâmica. Como o objetivo dos experimentos é validar a notação RPOO, dois dos três experimentos realizados apresentam estas características.

Com o objetivo de sistematizar os resultados obtidos, na apresentação de cada um dos três experimentos vamos considerar os seguintes aspectos: i) motivação da escolha; ii) objetivos do experimento; iii) descrição do sistema escolhido, com as suas características e propriedades; iv) os modelos obtidos; e, v) análise dos modelos, com as observações efetuadas durante o processo de construção dos produtos de modelagem, bem como o procedimento utilizado em sua validação.

Este capítulo está dividido em três seções, uma para cada experimento. Na Seção 3.1 apresentamos o experimento de modelagem do serviço *Bouncer*. Nas Seções 3.2 e 3.3 descrevemos a experimentação de RPOO no Sistema *DistBeta* e no protocolo *OSPF*, respectivamente.

3.1 Bouncer – Um Serviço Distribuído Para Controle de Licenças de Software

3.1.1 Motivação e Objetivos

O *Bouncer* é um serviço distribuído para o controle do uso de licenças de software em ambientes de redes locais [Bez96; BBF97]. Foi desenvolvido no Laboratório de Sistemas Distribuídos, do Departamento de Sistemas e Computação da Universidade Federal da Paraíba. Este serviço apresenta características de distribuição e concorrência, possuindo uma topologia de interconexão dinâmica.

O serviço *Bouncer* foi o primeiro sistema de software real, com diversas características e propriedades modelado em RPOO. Um dos fatores que contribuíram para esta escolha foi a facilidade de interação com a equipe que o concebeu e desenvolveu, possibilitando um completo entendimento do problema.

Um dos objetivos deste primeiro experimento foi contribuir para a definição de uma metodologia de desenvolvimento de modelagem com RPOO. Neste sentido, utilizamos os resultados obtidos por uma outra equipe que desenvolveu em paralelo um experimento de modelagem do mesmo problema, usando RPOO. Esta outra equipe foi constituída por dois alunos do curso de doutorado do departamento de Engenharia Elétrica desta Universidade, sendo que um dos integrantes é o responsável pela formalização da linguagem RPOO. A existência de dois modelos RPOO de um único sistema desenvolvido por equipes de modeladores diferentes, nos permite efetuar uma comparação entre os produtos de modelagem obtidos.

Um segundo objetivo deste experimento foi testar alguns construtos da linguagem, observando se estavam adequadamente definidos.

Durante o processo de modelagem objetivamos ainda identificar as dificuldades e facilidades oferecidas pela notação RPOO.

3.1.2 Descrição do Serviço Bouncer

O serviço *Bouncer* é um sistema distribuído para o controle do uso de licenças de software em ambientes de redes locais, permitindo que somente um número controlado de usuários

licenciados execute e utilize uma determinada aplicação simultaneamente. A política de licenciamento adotada pelo serviço *Bouncer* é a de quantidade de ativações concorrentes.

O serviço *Bouncer* adota um modelo hibrído de programação distribuída, mesclando os paradigmas ponto-a-ponto (P/P) e cliente-servidor (C/S). A comunicação C/S envolve um protocolo bastante simples, do tipo pedido-resposta, onde a resposta do servidor ao pedido do cliente funciona como uma confirmação do pedido enviado. No modelo P/P, uma mesma entidade acumula as funcionalidades de cliente e de servidor.

Na Figura 3.1 apresentamos um possível cenário do serviço *Bouncer*. Neste cenário, existem 02 máquinas *HOST1* e *HOST2*, conectadas através de um canal de comunicação. Em cada máquina da rede local em que existem processos clientes executando (aplicações que serão controladas pelo *Bouncer*), existe um único servidor *Bouncer*. Este é o caso, por exemplo, da máquina *HOST2* que possui dois processos clientes *PC1* e *PC2*, mas apenas o servidor Bouncer *SB2*.

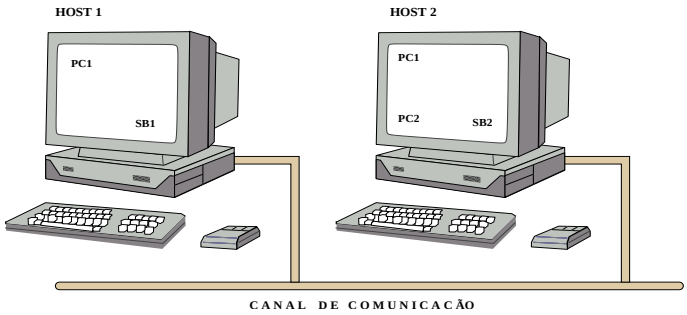


Figura 3.1: Cenário do Serviço *Bouncer*

O processo cliente é responsável por solicitar um servidor Bouncer ou inicializá-lo, caso ele não esteja ativo; solicitar uma licença para a sua execução e após o uso desta liberá-la.

Quando o servidor Bouncer é iniciado pelo processo cliente, sua primeira tarefa é ingressar no grupo bouncer. Caso o grupo não exista, o servidor irá criá-lo. Ao ingressar no grupo, o servidor receberá do líder do grupo uma tabela de licenças. O servidor Bouncer gerencia as licenças solicitadas pelos processos clientes. O procedimento de obtenção de licença inicia-se quando um processo cliente comunica-se diretamente com o servidor Bouncer de sua máquina, solicitando deste, uma licença para que possa continuar a sua execução. O servidor Bouncer, consulta a base de dados mantida por ele localmente para verificar quantas cópias da aplicação que solicitou a licença estão em execução em toda a rede. Cada servidor

ativo na rede envia informações sobre as licenças utilizadas em sua máquina para os demais servidores. Além disso, ele avisa também quando uma licença foi disponibilizada. Desta maneira, todas as tabelas dos servidores são consistentes entre si. Assim, o servidor Bouncer tem subsídios para decidir se concede ou não a licença para o processo cliente.

O serviço *Bouncer* possui dois pressupostos. Sempre que existir pelo menos um processo cliente em execução em uma máquina, um, e apenas um servidor Bouncer executando o serviço *Bouncer* deverá estar ativo naquela máquina; e antes de finalizar sua execução, o processo cliente libera a licença que ele detém.

O serviço *Bouncer* é executado por um grupo de comunicação formado pelos servidores Bouncers, denominado de grupo bouncer. Existem mecanismos para gerenciar este grupo, os quais devem permitir que o grupo possa ser criado, dissolvido, receber e liberar membros. Estes mecanismos são oferecidos pelo serviço de comunicação em grupo. Cada grupo bouncer possui um líder de grupo que é o seu membro mais antigo. Todas as mensagens enviadas ao grupo bouncer, deverão ser recebidas por todos os seus membros, inclusive o remetente, em uma mesma ordem.

3.1.3 Modelos RPOO

Nesta seção apresentamos os resultados da modelagem do serviço *Bouncer* utilizando a linguagem RPOO. Utilizamos a ferramenta Design/CPN [CJK97] para a edição do modelo. Optamos pelo uso desta ferramenta devido ao fato de termos familiaridade com ela e por possuir todos os recursos que precisávamos para editar os modelos, já que RPOO foi definida utilizando uma CP-Net para descrever o corpo das classes.

Como a linguagem RPOO não tem uma metodologia definida, optamos por utilizar uma das disciplinas de desenvolvimento de sistemas orientados a objetos. As etapas utilizadas para o desenvolvimento do experimento foram:

1. Identificação das classes do modelo, bem como suas respectivas associações.
2. Construção do diagrama de classes.
3. Identificação do diagrama de fluxo de mensagens – identificando todas as mensagens que podem ser trocadas entre os objetos das classes.

4. Elaboração do diagrama de configuração inicial – o qual contém todas as instâncias das classes, ligações entre os objetos e seus respectivos identificadores.

5. Detalhamento do corpo das classes do modelo.

Diagrama de Classes

Na Figura 3.2 é apresentado o diagrama de classes do modelo. As classes *HOST, CLIENT, BOUNCER* e *SERVICE GC* estão definidas porque enfatizam os principais componentes do sistema. Os nós do diagrama representam as classes e os arcos representam as associações. As inscrições nas extremidades dos arcos declaram a cardinalidade e os seletores, que são ligados ao identificador do objeto. Os seletores *host, clients*, *bouncer* e *service GC* serão utilizados no detalhamento das classes para efetuar a troca de mensagens. A notação utilizada pelo diagrama de classes RPOO segue em grande parte o estilo da notação UML [Lar98; RJB98].

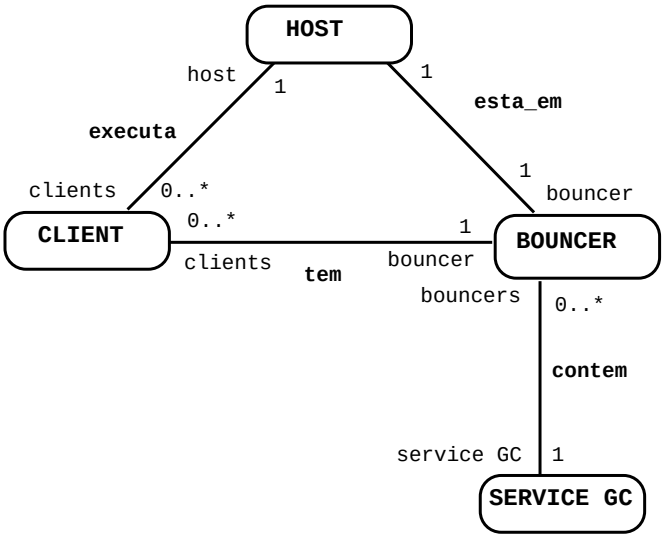


Figura 3.2: Diagrama de Classes

Diagrama de Fluxo de Mensagens

Na Figura 3.3 é apresentado o diagrama de fluxo de mensagens para o modelo. No diagrama é possível identificar todas as mensagens que podem ser trocadas entre os objetos do sistema. Como, por exemplo, entre as classes *CLIENT* e *BOUNCER*, podemos identificar todas as requisições que um objeto instanciado da classe *CLIENT* poderá fazer para um objeto

instanciado da classe *BOUNCER*, através das mensagens *req license* e *rel license* (requisição ou liberação de uma licença). Um objeto bouncer, por sua vez, enviará como resposta a uma requisição do objeto client as mensagens *lic ok*, *lic den* ou *lic released* (licença ok, licença negada ou licença liberada).

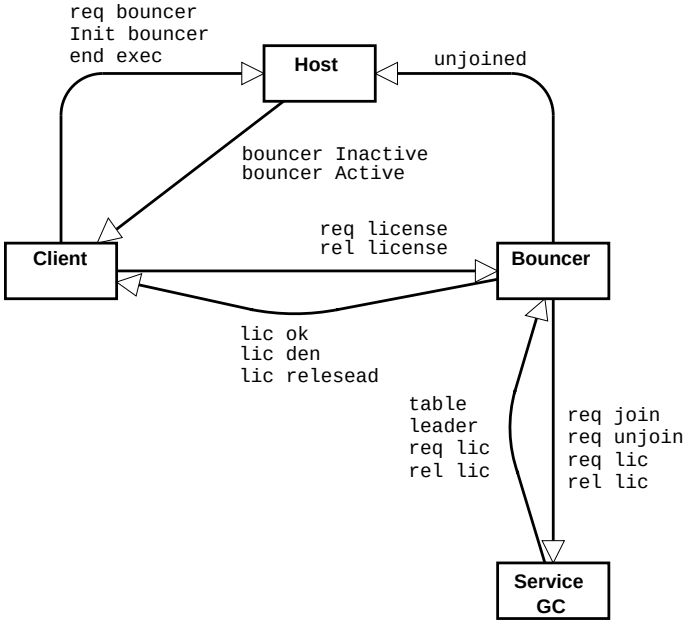


Figura 3.3: Diagrama de Fluxo de Mensagens

O diagrama de fluxo de mensagens pode ser gerado a partir do diagrama de classes e do detalhamento do corpo das classes. Contudo, ele foi descrito a priori, e foi utilizado como guia no detalhamento dos corpos das classes. Desta forma, este diagrama nos ajudou a manter consistente as trocas de mensagens efetuadas entre os objetos das classes.

Diagrama de Configuração Inicial

Na Figura 3.4 é apresentado um dos possíveis diagramas de configuração inicial para o modelo. Os nós representam os objetos e os arcos representam as ligações entre eles. De acordo com o diagrama, temos uma instância da classe *SERVICE GC* com o identificador *serviceGC*; duas instâncias da classe *CLIENT* com os identificadores: *client* e *client1*; uma instância da classe *BOUNCER* com o identificador *bouncer*, e uma instância da classe *HOST* com o identificador *host*.

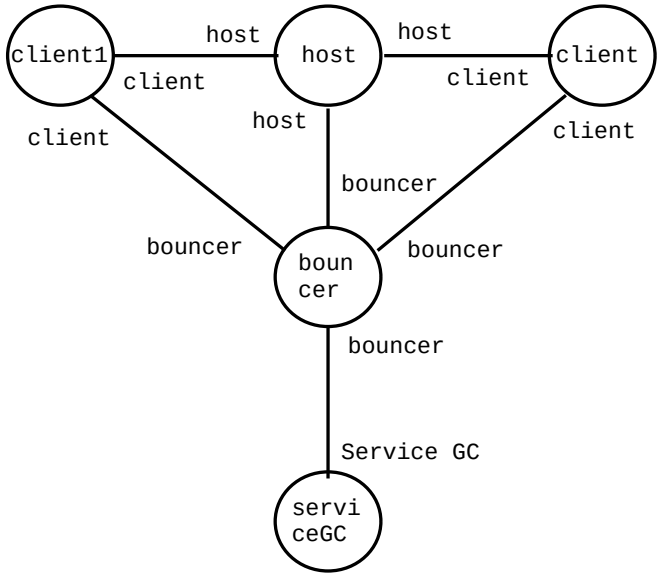


Figura 3.4: Diagrama de Configuração Inicial

Detalhamento das Classes

Classe HOST

Na Figura 3.5 é apresentado o corpo da classe *HOST*. O corpo da classe consiste em uma rede com dois lugares (*process client* e *server bouncer*) e cinco transições (*request bouncer, init_bouncer, request_bouncer, request end* e *request unjoin*). A classe *HOST* atende às requisições dos objetos das classes *CLIENT* e *BOUNCER*. A transição *request bouncer* tem duas inscrições de interação: *client?req bouncer*, que é uma inscrição de interação de entrada; e *client!bouncer Inactive*, que é uma inscrição de interação de saída síncrona. A ocorrência desta transição indica que o host recebeu uma mensagem do client solicitando um servidor Bouncer para que ele possa executar. O host enviará como resposta uma mensagem para o client informando que o servidor Bouncer está inativo.

Ao receber a mensagem do host informando que o servidor bouncer está inativo, o client enviará ao host a mensagem *Init bouncer*. Quando o host receber esta mensagem a transição *init_bouncer* irá ocorrer. Esta transição tem as seguintes inscrições de interação: *client?Init bouncer*, indicando que o host recebeu uma mensagem do client solicitando a iniciação de um servidor Bouncer; *bouncer = new BOUNCER*, que é uma inscrição de interação de instanciação, ou seja, criação do bouncer; e *client.bouncer Active,bouncer*, indicando que o host

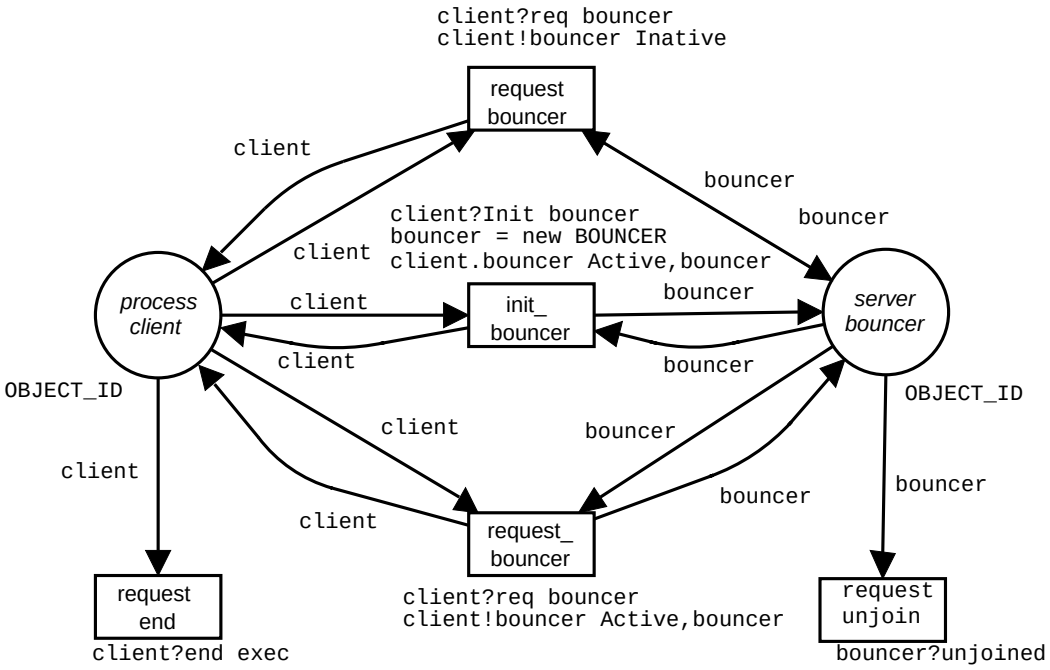


Figura 3.5: Classe HOST

está enviando uma mensagem para o client informando que o servidor Bouncer foi iniciado e qual é o seu identificador. A transição *request_bouncer* modela a situação na qual o host recebe a requisição de um servidor Bouncer do client e o bouncer já está ativo.

A ocorrência das transições *request unjoin* e *request end* indicam que o host recebeu a mensagem *bouncer?unjoined* do bouncer, informando a finalização da sua execução e a mensagem *client?end exec* do client informando que ele finalizou a sua execução.

Classe CLIENT

Na Figura 3.6 é apresentado o corpo da classe *CLIENT*. O corpo da classe consiste em uma rede com seis lugares (*process client exec, initializing sb, process client with sb, waiting requested lic, process client with lic* e *waiting release*) e oito transições (*request bouncer, request_bouncer, get bouncer, request license, get lic denied, get lic ok, request release* e *get released*). A marcação da rede determina o estado inicial dos objetos instanciados desta classe. Logo, um client tem como estado inicial a marcação **(Client,1)** no lugar *process client exec*, indicando que inicialmente o processo cliente está pronto para requisitar por um

bouncer.

A partir do estado *process client exec*, podem ocorrer as transições *request bouncer* e *request_bouncer*. A ocorrência destas transições indicam a situação em que um client solicita ao host um servidor Bouncer para que ele possa executar. Caso exista um servidor Bouncer ativo no host, o seu identificador será informado ao client. Caso o servidor Bouncer esteja inativo, o client irá solicitar ao host a sua iniciação, como podemos observar nas inscrições de interação das referidas transições.

Quando o processo cliente conhecer o servidor Bouncer de sua máquina (estado em que existe uma ficha no lugar *process client with sb*), ele estará apto a requisitar uma licença ao bouncer para continuar a sua execução, enviando a mensagem *bouncer.req license*(*transição request license*). O client ficará aguardando a resposta da requisição de licença.

A partir do estado *waiting requested lic* podem ocorrer duas transições. Ocorrendo a transição *get lic ok*, o client receberá a mensagem *bouncer?lic ok*, indicando que poderá continuar a sua execução e uma ficha será depositada no lugar *process client with lic*. Ocorrendo a transição *get lic denied*, o client receberá a mensagem *bouncer?lic den* do bouncer informando que a licença para a sua execução foi negada. Ao receber esta mensagem, o client irá enviar a mensagem de auto-destruição *host!end exec* ■ para o host.

Quando o processo cliente terminar a sua execução, uma mensagem requisitando a liberação da licença que ele detém será enviada ao bouncer (transição *request release*). Ao receber a mensagem *bouncer?lic released* confirmando a liberação da licença, o client enviará uma mensagem de auto-destruição *host!end exec* ■ para o host, informando que está encerrando a sua execução (transição *get released*).

Classe BOUNCER

Na Figura 3.7 é apresentado o corpo da classe *BOUNCER*. O corpo da classe consiste em uma rede com dois lugares (*server bouncer* e *sb with table*) e sete transições (*request join, request license, release license, get_lic ok, get_lic release, get_lic denied* e *request unjoin*).

Quando o servidor Bouncer é iniciado pelo processo cliente, ele segue um processo de iniciação para atualizar a sua base de dados de acordo com a base de dados dos demais bouncers do grupo bouncer. Este processo é modelado pela transição *request join*.

Ao disparar a transição *request join* o bouncer envia a mensagem *service GC!req join*

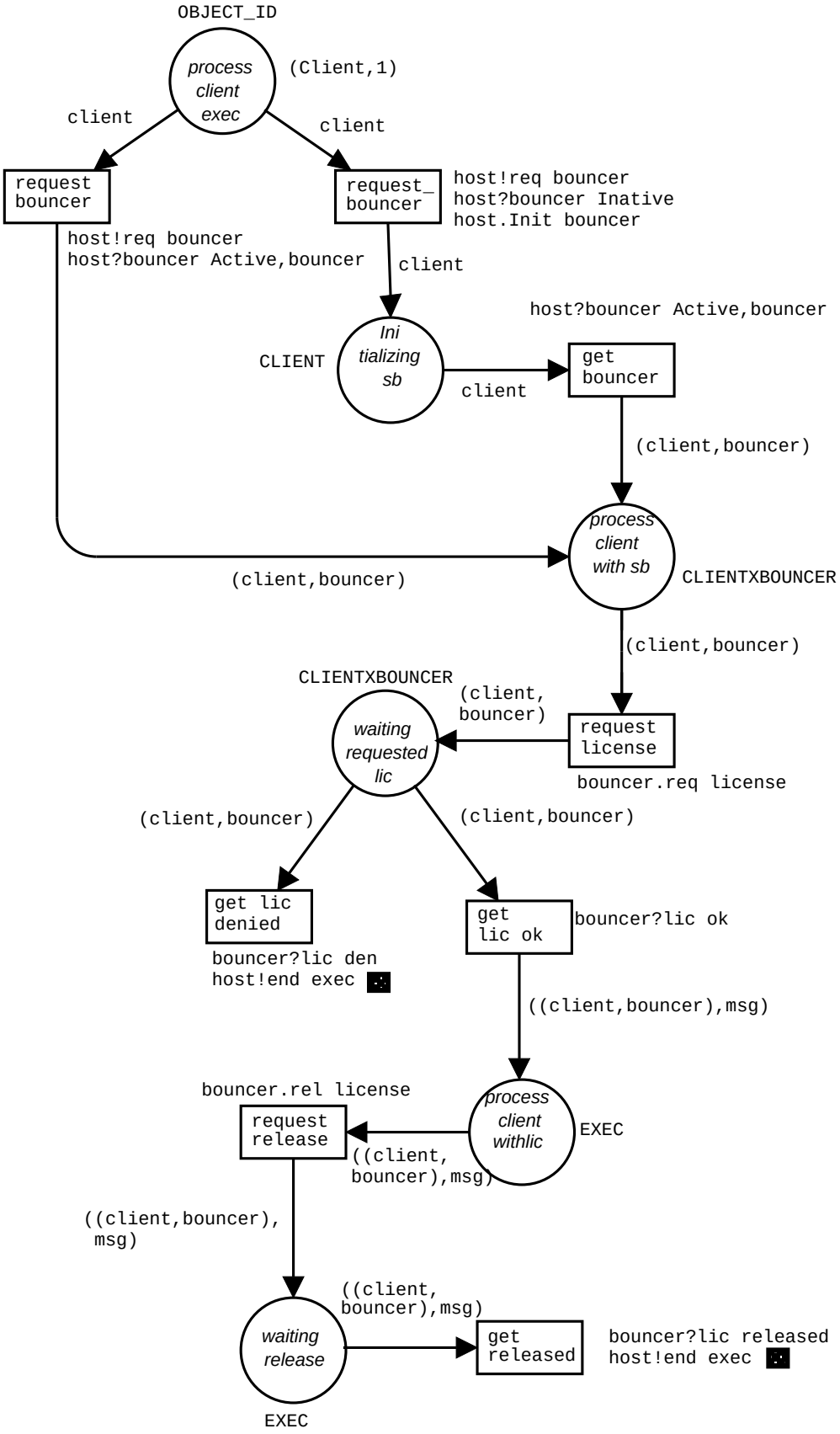


Figura 3.6: Classe CLIENT

para o service GC (que modela o serviço de comunicação em grupo), requisitando a sua entrada no grupo bouncer. Ao enviar esta mensagem o bouncer receberá como resposta do service GC a mensagem *service GC?lid*, informando a identificação do líder do grupo bouncer e a mensagem *service GC?t* atualizando a sua base de dados. Em seguida, o bouncer estará apto a atender os pedidos de requisição e liberação de licenças do client, estado em que existe uma ficha no lugar *sb with table*.

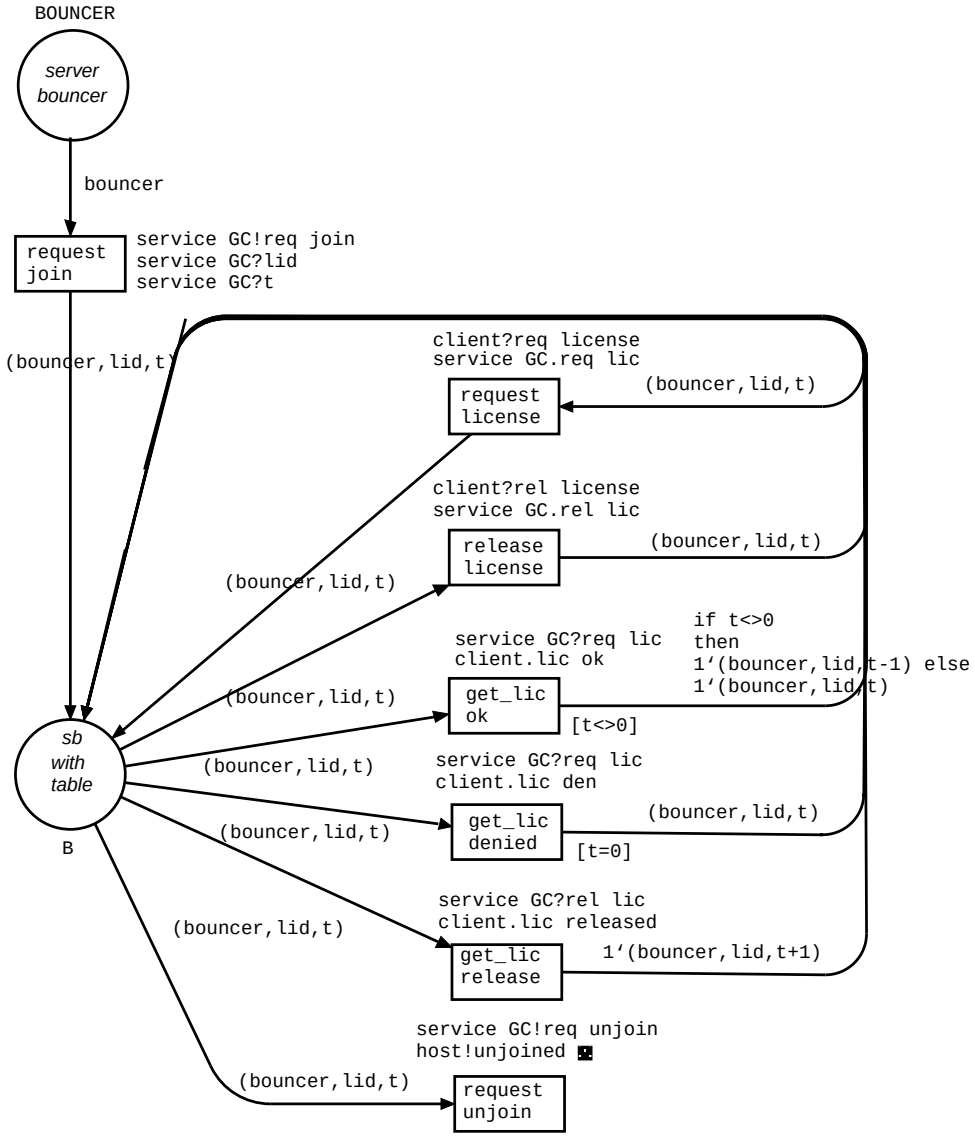


Figura 3.7: Classe BOUNCER

Quando as transições *request license* e *release license* ocorrerem, as requisições de serviços recebidas através das mensagens *client?req license e client?rel license* serão repassadas para o grupo bouncer (objeto service GC), através das mensagens *service GC.req lic* e *ser-*

vice GC.rel lic. Ao receber a mensagem *service GC?req lic* do service GC, o bouncer irá responder as resquisições do client de acordo com a sua base de dados. Caso exista licença disponível, a transição *get_lic ok* irá ocorrer e a mensagem *client.lic ok* será enviada ao client. Se não houver licença disponível, a transição *get_lic denied* irá ocorrer e a mensagem *client.lic den* será enviada ao client, informando que o pedido de licença foi negado.

A transição *get_lic release* modela a situação na qual o bouncer recebe a mensagem *service GC?rel lic* do service GC, e envia a mensagem *client.lic released* para o client confirmando o pedido de liberação de licença.

Quando o servidor Bouncer não estiver gerenciando a execução de nenhum processo cliente, ele irá enviar a mensagem *service GC!req unjoin* para o service GC requisitando a sua retirada do grupo bouncer e a mensagem *host!unjoined* ■ para o host finalizando a sua execução (*transição request unjoin*).

Classe SERVICE GC

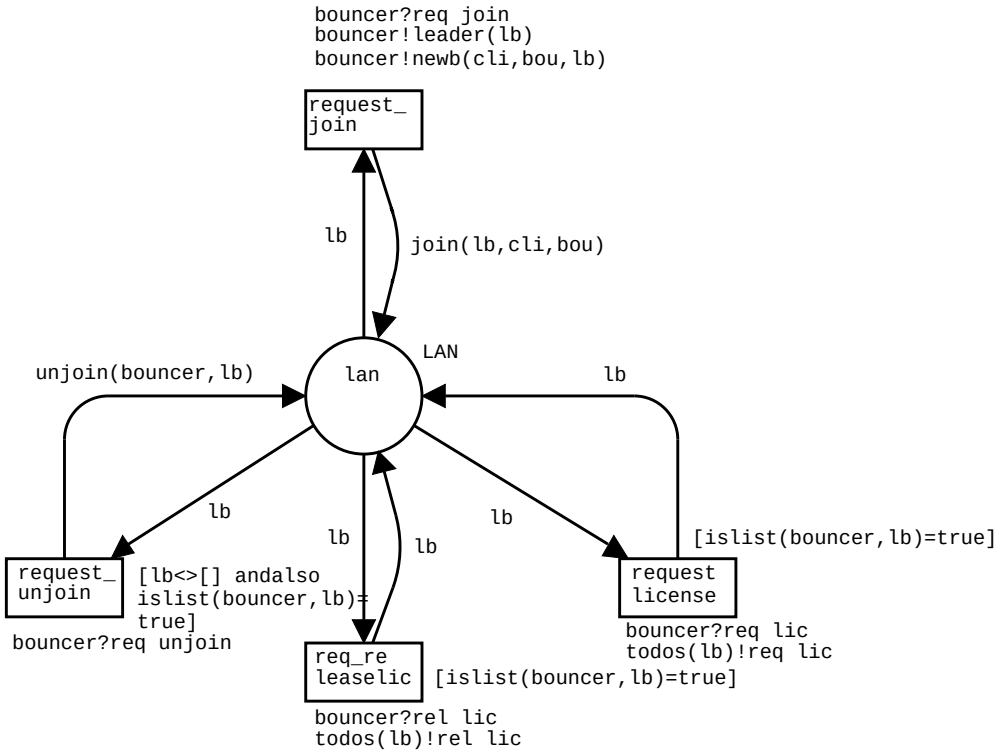


Figura 3.8: Classe SERVICE GC – Serviço de Comunicação em Grupo

Na Figura 3.8 é apresentado o corpo da classe *SERVICE GC*. O corpo da classe con-

siste em uma rede com um lugar *lan* e quatro transições (*request_join, request license, req_release lic* e *request_unjoin*). Quando o service GC receber do bouncer a mensagem *bouncer?req join* (transição *request_join*), ele enviará as mensagens *bouncer!leader(lb)* e *bouncer!newb(cli,bou,lb)* para o bouncer informando o seu líder de grupo e a sua base de dados.

As transições *request license* e *req_release lic* modelam o recebimento das mensagens de requisição e liberação de licenças *bouncer?req lic* e *bouncer?rel lic* de um bouncer. Ao receber estas mensagens o service GC simplesmente as repassará para todos os membros do grupo bouncer, através das mensagens *todos(lb)!req lic* e *todos(lb)!rel lic*. A função *todos(lb)* ao ser avaliada retorna todos os bouncers pertencentes ao grupo bouncer.

Quando o service GC receber a mensagem *bouncer?req unjoin* de um bouncer (transição *request_unjoin*), ele removerá o referido objeto do grupo bouncer.

3.1.4 Análise do Experimento

Validação do Modelo RPOO

Para efetuarmos a validação do modelo, utilizamos o sistema para simulação de RPOO. Este sistema é formado por um conjunto de ferramentas integradas que oferecem suporte computacional à simulação de modelos RPOO [San01]. A arquitetura do sistema é composta basicamente por três elementos: um simulador para o sistema de objetos; um simulador para redes de Petri; e finalmente, um elemento responsável por executar a comunicação entre os simuladores do sistema de objetos e das redes de Petri. A Figura 3.9 representa a arquitetura do simulador de RPOO.

O *Simulador de Sistema de Objetos* (SSO) é uma ferramenta que permite simular as alterações ocorridas em um sistema de objetos através da execução de ações. A ferramenta SSO pode ser considerada como um mediador, que trata as requisições de execução de ações de várias instâncias de objetos, validando ou não as ações requeridas. O comportamento dos objetos que se comunicam com o SSO é representado por uma CP-Net.

O SSO é uma ferramenta que permite a criação de uma estrutura inicial para o sistema de objetos e a criação de eventos sobre essa estrutura. A ferramenta possui uma representação interna para as estruturas, os eventos e as regras que definem os efeitos dos eventos sobre

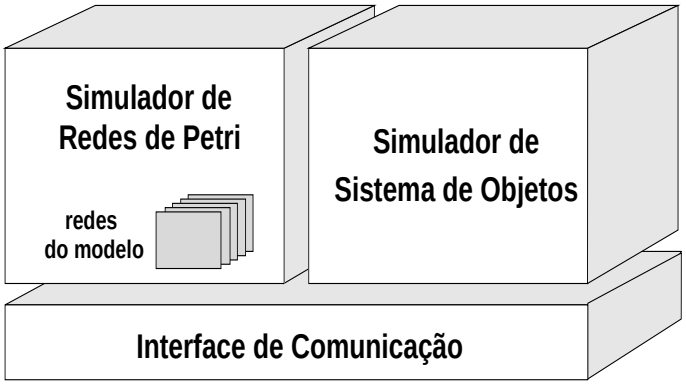


Figura 3.9: Arquitetura do Simulador de RPOO

uma estrutura do sistema de objetos. A partir destes elementos, o SSO permite simular o efeito de qualquer evento sobre determinada estrutura, ou seja, é possível verificar qual a estrutura resultante em relação a um evento aplicado sobre uma estrutura inicial.

O *Simulador de Redes de Petri* é o módulo que representa a visão das redes de Petri de um modelo. Neste módulo utilizamos a ferramenta Design/CPN [CJK97] ou, mais especificamente, o módulo de simulação do Design/CPN.

O módulo *Interface de Comunicação*—Figura 3.9, representa a comunicação entre os módulos *Simulador de Redes de Petri* e *Simulador de Sistema de Objetos*. A integração entre esses dois módulos não foi automatizada ainda. Assim, a entrada de dados na ferramenta SSO deve ser feita pelo usuário, a partir de requisições através de linhas de comando ou de arquivos. Com o SSO é possível simular o comportamento de um sistema de objetos modelado em RPOO. Para utilizar o SSO, existe uma gramática definida e uma linguagem para entrada de dados no sistema, que utilizam a representação algébrica proposta em RPOO [Gue02]. Qualquer expressão algébrica enviada para o SSO é interpretada como uma estrutura. Inicialmente, o SSO deve receber a estrutura que representa a configuração inicial do modelo RPOO; posteriormente os eventos/ações que serão disparados e que podem ou não modificar esta estrutura. A Figura 3.10 apresenta as possíveis ações que podem ser efetuadas no SSO.

Durante a validação do modelo, desempenhamos o papel do elemento responsável por executar a comunicação entre os simuladores do sistema de objetos e das redes de Petri. Os resultados da simulação do serviço *Bouncer*, foram analisados a partir dos relatórios gerados automaticamente pela ferramenta Design/CPN após a simulação de cada objeto do sistema.

Cada passo registrado durante a simulação das redes de Petri no Design/CPN, foi conver-

Evento/Ação	Nome
+ x	Ação local (ou interna) criação ou instanciação de objetos
x?m	Entrada de dados
x.m	Saída assíncrona de dados
x!m	Saída síncrona de dados
– x	Desligamento ou remoção de ligação ação final (ou auto−destruição)

Figura 3.10: Ações Elementares do SSO

tido manualmente em uma mensagem para o SSO. A validação da ação pelo SSO, por sua vez, foi utilizada para indicar qual o conjunto de ações, ou transições, que poderia/deveria ser disparado pelas redes de Petri. Embora a integração tenha sido feita manualmente, a simulação aumentou o grau de confiabilidade na modelagem do problema e ofereceu uma importante contribuição para a avaliação da notação, porque permitiu a análise do comportamento dos modelos que utilizam a formalização de RPOO. A etapa de simulação contribuiu na identificação de alguns erros nas funções utilizadas no modelo, os quais foram corrigidos durante a simulação. Por exemplo, em algumas funções os nomes das constantes estavam definidos de uma maneira, *val status=getStatus(Bid)*, e na comparação dos parâmetros eram utilizados nomes diferentes, *if sta=“active”*.

Para efetuar a validação do modelo consideramos diversos cenários. A Figura 3.4 mostra uma das configurações utilizadas. Esta configuração consiste em um host, um bouncer, um service GC e dois processos client. Para visualizar as ações ocorridas nos objetos instanciados e as mensagens trocadas entre eles, utilizamos o diagrama de sequência de mensagem (MSC) [MSC] gerado automaticamente durante as simulações efetuadas no Design/CPN.

Para cada classe do modelo RPOO foi gerado um gráfico *MSC* com as ações ocorridas nos objetos instanciados. As ações estão rotuladas com as mensagens trocadas entre os objetos, que são as mesmas das inscrições de interação associadas às transições das redes de Petri que modelam os corpos das classes. A Figura 3.11, representa a sequência de eventos que foram gerados pelo MSC. Em cada MSC representamos somente os objetos que se comunicam com a instância em questão. As Figuras 3.12, 3.13, 3.14, 3.15 e 3.16, representam

graficamente os MSCs gerados. As colunas *Client* e *Client2* representam os objetos *client* instanciados; a coluna *Host* representa o objeto *host*; a coluna *Bouncer* representa o objeto *bouncer* e a coluna *ServiceGC* representa o objeto *serviceGC*. As mensagens enviadas são representadas no MSC por um arco direcionado, definindo o objeto emissor e receptor. As setas são rotuladas com o tipo da mensagem enviada/recebida. Um processamento interno no objeto é representado no MSC com uma marca ■ na coluna correspondente ao objeto.

Número do Evento	Evento	Objeto
1	Requisição de um bouncer (req bouncer)	Client
2	Inicialização de um bouncer (Init bouncer)	Host
3	Requisição de ingresso no grupo bouncer (req join), líder e tabela.	Bouncer
4	Requisição de licença ao servidor bouncer−Bouncer (req license)	Client
5	Repasse da requisição de licença ao serviço de comunicação em grupo−ServiceGC (req lic)	Bouncer
6	Respondendo a requisição de de licença	Service GC
7	Recebendo a requisição de licença (licença OK : lic ok)	Client
8	Requisição de um bouncer (req bouncer)	Client
9	Requisição de licença ao servidor bouncer (req license)	Client
10	Repasse da requisição de licença ao serviço de comunicação em grupo−ServiceGC (req lic)	Bouncer
11	Requisição de Liberação de licença ao servidor bouncer (rel license)	Client
12	Repasse da requisição de liberação de licença ao grupo−ServiceGC (rel lic)	Bouncer
13	Respondendo a requisição de licença (licença negada : lic den)	Bouncer
14	Respondendo a liberação de licença (lic released)	Bouncer
15	Client finalizando a execução (end exec)	Client
16	Bouncer finalizando a execução (req unjoined)	Bouncer

Figura 3.11: Sequência de Eventos do SSO

Os rótulos com inscrição *Evento* indicam o evento que é executado no SSO. Assim, o

Evento 1, das Figuras 3.12 e 3.13, representa o evento:

Client1:host!req bouncer @

host:Client1?req bouncer @

host:Client1!bouncer Inactive @

Client1:host?bouncer Inactive @

Client1:host.Init bouncer, executado pelo SSO. O símbolo @ é utilizado para representar a composição de ações em um evento.

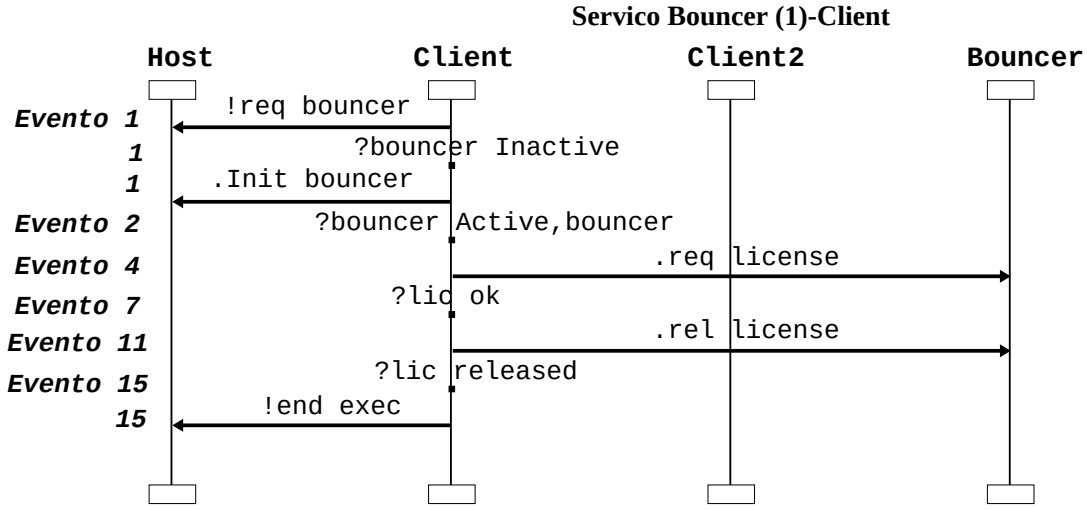


Figura 3.12: Gráfico MSC do Objeto *Client1*

A inscrição de interação *!req bouncer* (coluna Client da Figura 3.12), indica que o objeto client (origem) está enviando a mensagem *req bouncer* de maneira síncrona para o objeto host (destino). A inscrição *?req bouncer* (coluna Host da Figura 3.13) indica que o objeto host recebeu (executou uma ação ■) a mensagem *req bouncer* enviada pelo objeto client. Como podemos observar na coluna Host do MSC, quando o objeto host executa o processamento desta mensagem, ele envia uma mensagem de volta ao objeto client informando se o Servidor Bouncer está ativo ou não. Neste caso, a resposta foi negativa, mensagem *!bouncer Inactive*. Como podemos observar na coluna *Client* da figura 3.12, quando o objeto client recebe a mensagem *!bouncer Inactive* enviada pelo objeto host, ele solicita a iniciação do servidor Bouncer ao referido objeto, através da mensagem *.Init bouncer*. De acordo com a especificação do serviço *Bouncer*, o objeto client é responsável por iniciar o *Bouncer*, caso este esteja inativo.

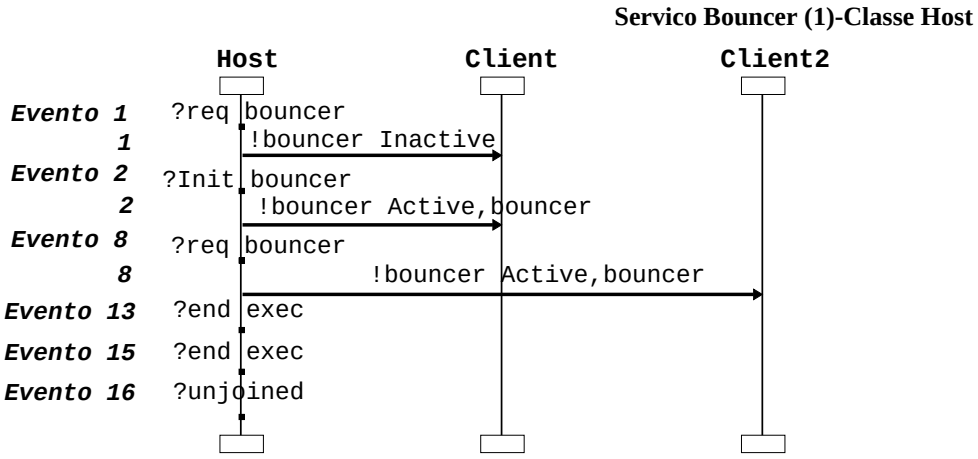


Figura 3.13: Gráfico MSC do Objeto *Host*

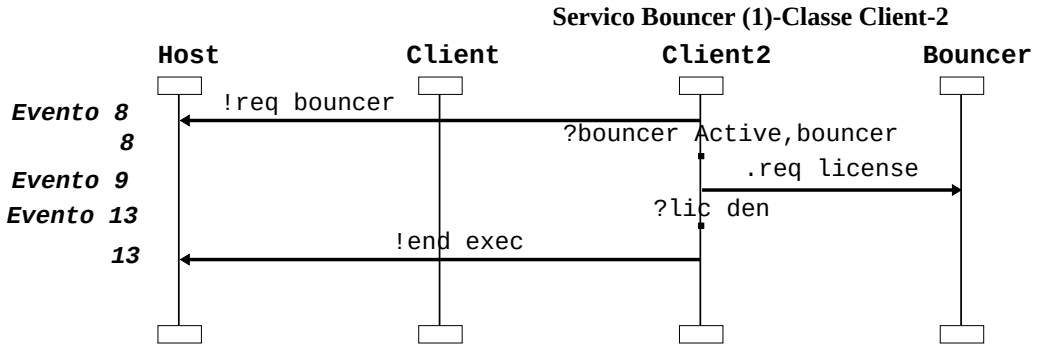


Figura 3.14: Gráfico MSC do Objeto *Client2*

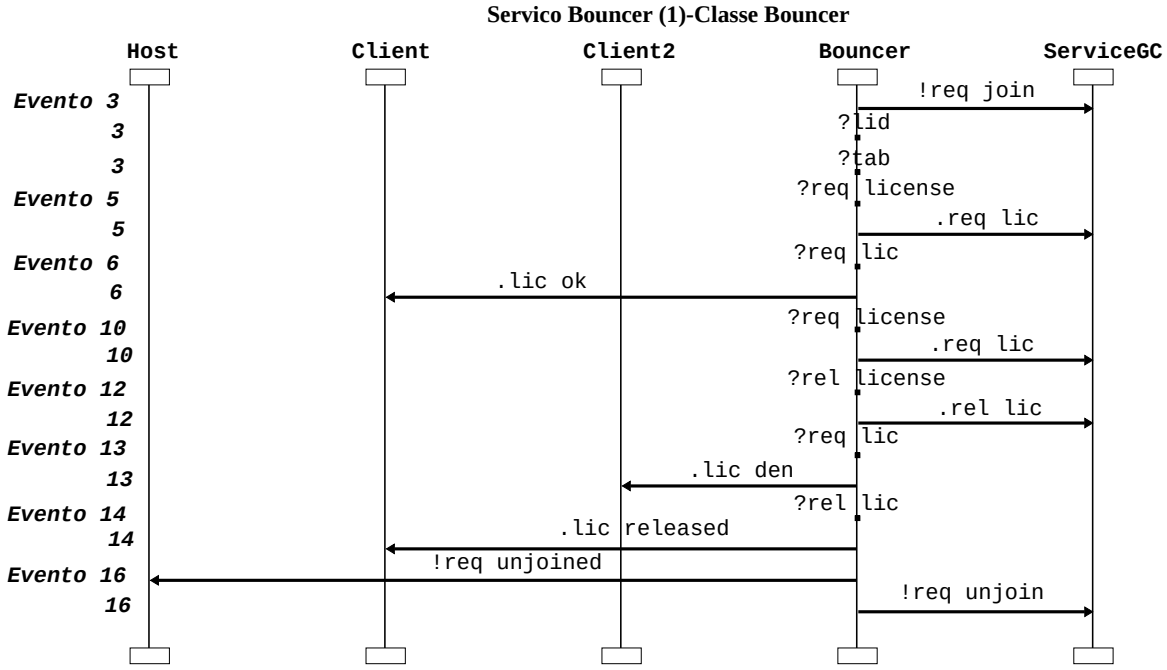


Figura 3.15: Gráfico MSC do Objeto *Bouncer*

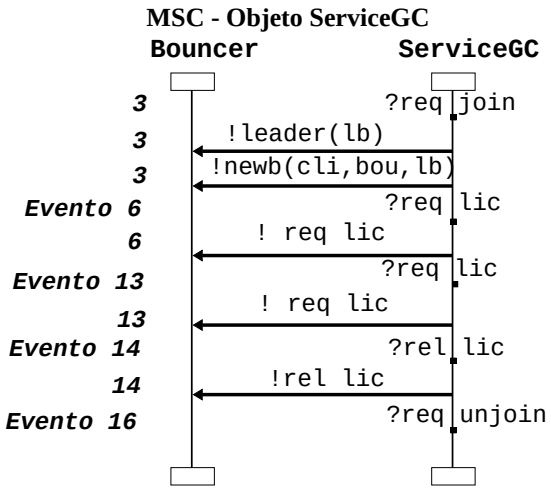


Figura 3.16: Gráfico MSC do Objeto Service GC

Quando o objeto bouncer recebe a mensagem *?req license* do objeto client (coluna Bouncer da Figura 3.15), ele repassa essa mensagem (*.req lic*) ao objeto ServiceGC. O objeto bouncer somente responderá a uma requisição de licença do objeto client após receber a mensagem *.req lic* do objeto ServiceGC. As mensagens *!req unjoined* e *!req unjoin* representam a destruição do servidor *Bouncer*, pois neste momento não existe nenhum objeto client executando. Quando o objeto ServiceGC receber a mensagem *?req unjoin* ele irá remover o objeto bouncer do grupo bouncer (coluna ServiceGC da Figura 3.16).

O processo de validação do modelo utilizando a ferramenta RPOO nos dá confiança de que o modelo está correto. As trocas de mensagens entre os objetos foram efetuadas da maneira como esperávamos (os resultados da simulação usando o SSO foram armazenados no arquivo *simulacao.log*, onde verificamos que após cada ação(evento) sobre a estrutura vigente, a estrutura resultante era a esperada). Ou seja, as mensagens foram enviadas e recebidas corretamente pelos objetos do modelo. Além disso, os resultados das simulações efetuadas de cada classe isoladamente pelo Design/CPN, verificando o comportamento dos objetos em diferentes situações nos dá confiança na corretude dos modelos das classes, pois durante a simulação o comportamento dos objetos instanciados foram exatamente os definidos pela especificação do serviço *Bouncer*.

Impressões Sobre a Solução Obtida

Mesmo considerando a pouca experiência em modelagem com a notação RPOO, podemos afirmar que o processo de modelagem foi rápido, comparando com experiências prévias de desenvolvimento de modelagem usando redes de Petri. Conhencimento de orientação a objetos e experiência em modelagem com redes de Petri coloridas facilitaram o trabalho de modelagem.

O modelo obtido é de fácil entendimento. O diagrama de classes do modelo nos fornece uma visão geral do sistema, permitindo identificar claramente todos os objetos e seus relacionamentos. A linguagem utilizada no diagrama de classes é convencional para toda a orientação a objetos. A vantagem da linguagem RPOO é a aderência aos conceitos de orientação a objetos e redes de Petri que já existem. O comportamento de uma classe RPOO é detalhada em uma única rede. No detalhamento das classes *HOST, CLIENT, BOUNCER* e *SERVICE GC*, podemos identificar facilmente o comportamento dos objetos e as trocas de mensagens efetuadas entre eles, através das inscrições de interação associadas às transições da rede. Em RPOO as mensagens são enviadas para o identificador dos objetos, de acordo com as ligações usadas pela Orientação Objetos.

A comunicação entre as partes do sistema é modelada de forma bem simples em RPOO. As trocas de mensagens para a solicitação e prestação de serviços são efetuadas através das inscrições de interação associadas às transições da rede, simplificando o envio de mensagens aos objetos do sistema. Assim, tivemos uma grande facilidade em modelar as requisições de serviços.

Dentre as dificuldades encontradas está a falta de uma ferramenta mais completa de apoio computacional para análise e verificação dos modelos desenvolvidos.

Contribuições do Experimento

Como mencionado anteriormente, a modelagem do Bouncer seguiu as etapas tradicionais de um processo de desenvolvimento de sistemas orientado a objetos. Esse parece ser um caminho natural, especialmente se a equipe de desenvolvimento tem experiência na utilização de conceitos de orientação a objetos. Comparando os produtos de modelagem do Serviço *Bouncer* utilizando a linguagem RPOO desenvolvidos pela outra equipe de modeladores com os

nossos produtos obtidos, observamos que identificamos as mesmas classes e o detalhamento do corpo destas classes estão bem parecidos. Como em RPOO a decomposição é feita pelos objetos do sistema, a identificação das classes coincidiu, pois os objetos do sistema são únicos para qualquer decomposição. A forma como os conceitos da orientação a objetos foram integrados com redes de petri impõem uma certa disciplina orientada a objetos para construção de modelos RPOO.

É possível, portanto, identificar as seguintes etapas na modelagem de sistemas RPOO:

1. **identificação dos objetos do sistema** – é importante identificar todos os objetos do modelo inicialmente, porque assim podemos separá-los de acordo com as suas atividades desenvolvidas no sistema;

2. **identificação das classes do modelo e suas associações** – ao identificarmos as classes e como elas estão associadas, podemos separar os objetos de acordo com o seu comportamento e ligações no sistema;

3. **elaboração do diagrama de classes** – quando definimos o diagrama de classes, temos uma visão abstrata dos objetos do sistema e de seus relacionamentos;

4. **elaboração do diagrama de fluxo de mensagens** – com a sua elaboração identificamos todas as mensagens que podem ser trocadas entre os objetos do sistema, o que nos ajudará a manter consistente as partes do modelo;

5. **elaboração do diagrama de configuração inicial** – com a sua elaboração indicamos o número de instâncias de cada objeto que existe no sistema no momento da sua iniciação;

6. **detalhamento do corpo das classes** – ao detalharmos o corpo das classes após as etapas anteriores, teremos identificado todos os possíveis comportamentos dos objetos do sistema, o que facilita a modelagem de cada um deles.

Um outro resultado importante do experimento foi a aplicação da sintaxe da linguagem RPOO. A sintaxe que define o envio simultâneo de mensagens para vários objetos foi modificada. A semântica da linguagem permitia o envio simultâneo, mas a maneira como a sintaxe

estava definida restringia o envio simultâneo de mensagens. Na definição anterior, era necessário enumerar as variáveis que correspondiam aos identificadores dos objetos para os quais a mensagem estava sendo enviada. Por exemplo, para enviar uma mensagem síncrona para os objetos *a*,*b* e *c* requisitando um servidor bouncer : *a!req bouncer*, *b!req bouncer* e *c!req bouncer*. A nova definição desta sintaxe utiliza uma função na inscrição de interação, que ao ser avaliada resulta em uma lista de objetos. Assim, a mensagem seria: *todos(lb)!req bouncer*. Isto equivale a poder determinar dinamicamente os objetos que devem receber as mensagens. Observe que isto também significa que as classes a que pertencem os objetos só são conhecidas em tempo de execução/simulação.

Com o uso das ferramentas SSO e Design/CPN efetuamos uma verificação informal do modelo RPOO. Com o desenvolvimento de uma ferramenta completa de suporte computacional para a modelagem, análise e verificação de modelos RPOO poderemos submeter os produtos desenvolvidos a uma verificação formal.

3.2 Execução de Programas Distribuídos em BETA (DistBeta)

3.2.1 Motivação e Objetivos

O segundo experimento de modelagem escolhido foi o sistema *DistBeta*, um framework para execução de programas distribuídos desenvolvido na linguagem de programação orientada a objeto BETA [JM95; Jen92a].

A maior motivação para esta escolha foi o fato de existir uma especificação formal desse sistema utilizando redes de Petri coloridas [Jen92c]. A existência de uma especificação formal, utilizando uma notação que é bastante usada em nosso grupo de pesquisa, permitiu o completo entendimento do sistema, mesmo sem a participação de membros da equipe de desenvolvedores do *DistBeta*. A construção dos produtos de modelagem partiram da análise desta especificação.

A modelagem foi desenvolvida por um aluno de mestrado da COPIN, sob a minha supervisão. O aluno tinha experiência na modelagem de sistemas usando redes de Petri coloridas e conhecimento de orientação a objetos. O objetivo maior deste experimento foi verificar a

utilização de RPOO por usuários novatos, sem experiência no formalismo.

Durante o estudo do experimento, identificamos a metodologia utilizada pelo modelador, bem como as dificuldades e facilidades encontradas na utilização de RPOO.

3.2.2 O DistBeta

A Figura 3.17 apresenta uma visão geral da arquitetura do sistema *DistBeta*. É possível identificar três componentes básicos: *Ensemble, Shell* e *Thread*. O *Ensemble* é uma representação do sistema operacional em uma máquina conectada a rede. Podemos considerar o *ensemble* como um *host*. O conceito de *Shell* é semelhante ao conceito de um processo. Os *Shells* existem dentro dos *Ensembles* e são mantidos por eles. O *Shell* pode comunicar-se com outros *Shells* ou *Ensembles*. Podem existir vários *Shells* configurados em um *Ensemble*. O *Shell* contém uma tabela com os identificadores dos objetos locais e objetos remotos disponíveis no sistema.

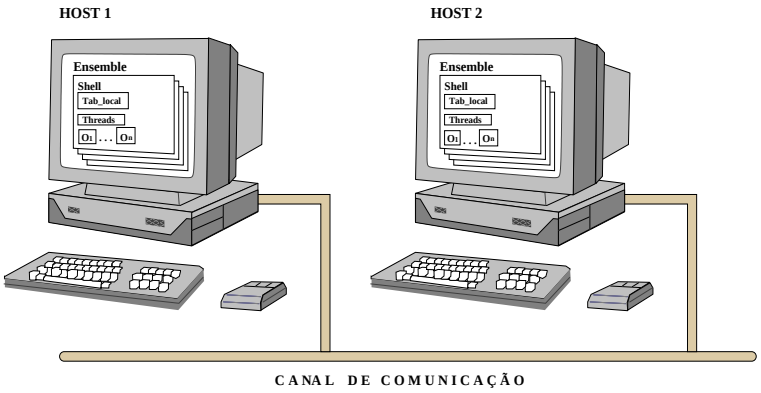


Figura 3.17: Sistema DistBeta

Os *Threads* são responsáveis pela execução das tarefas dentro dos *Shells*. Existem três tipos de *threads*:

1. *User thread*: responsável por iniciar a comunicação entre os objetos, atendendo as requisições feitas pelos objetos. Cada *Shell* contém pelo menos um *user thread* que executa o programa principal.

2. *Listener thread*: responsável por atender as requisições que chegam ao *Shell* provenientes da rede. Existe apenas um *listener thread* para cada *Shell*.

3. *Worker thread*: responsável por fazer com que um objeto execute a tarefa que lhe foi requisitada. O *worker thread* é criado pelo *listener thread* do *shell* sempre que existir uma requisição a ser executada.

Na figura 3.17 temos uma configuração com dois *hosts* que pertencem a uma mesma rede. Em um sistema *DistBeta* vários *Hosts* podem estar conectados a uma mesma rede. A comunicação entre objetos é efetuada através de chamadas remotas de objetos. Suponha que um objeto *O1*, instanciado no *Host1*, deseja comunicar-se com um objeto *O2*, instanciado no *Host2*. A comunicação entre *O1* e *O2* pode ser descrita em 06 etapas (Figura 3.18):

1. O objeto *O1* solicita o identificador (OID) do objeto *O2* em uma tabela local ao *Shell*, contendo os OIDs dos objetos remotos. Um *user thread* é alocado para esta tarefa. O *user thread* contém os métodos necessários para serializar[1] os parâmetros para que a mensagem possa ser enviada através de um método de chamada remota, como um RPC (Procedimento de Chamada Remota).

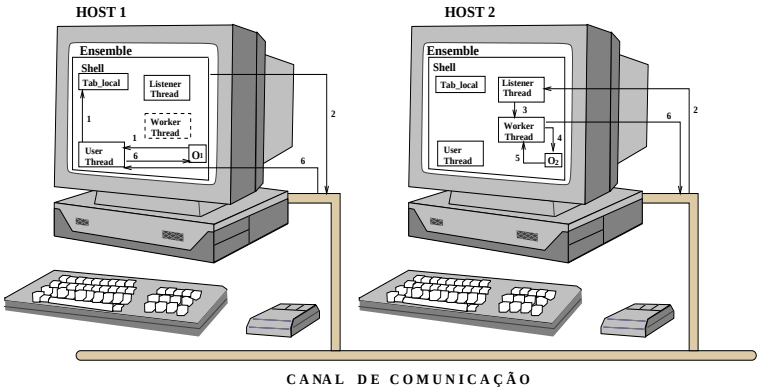


Figura 3.18: Comunicação entre dois Objetos

2. A mensagem é enviada para o *Host2* e o objeto *O1* é bloqueado.

3. O *listener thread* no *Shell* do *Host2* recebe a requisição e aloca um *worker thread*, passando a mensagem para que ele se responsabilize pela execução da tarefa. Ao alocar o *worker thread*, o *listener thread* estará apto a receber novas requisições.

4. O *worker thread* desserializa a mensagem[2], identificando o objeto que deve ser chamado e solicita ao objeto *O2* a execução da tarefa solicitada pelo objeto *O1*, enviando

[1]Transformar a mensagem em um fluxo de bytes para que possa ser enviada através da rede.

[2]Transforma o fluxo de bytes proveniente da rede na mensagem que foi enviada pelo *shell* no *Host1*.

os parâmetros necessários.

5. O objeto *O2* executa a tarefa e envia o resultado ao *worker thread*, que vai serializar novamente os parâmetros para retornar a mensagem para o objeto *O1*. O resultado serializado é enviado para o *Host1* e o *worker thread* é liberado.

6. O *user thread* do *Shell* no *Host1* vai receber a mensagem (o qual estava bloqueado, aguardando o recebimento da mensagem), desserializá-la e entregá-la ao objeto *O1*. O objeto *O1* e o *user thread* são liberados.

A sequência de eventos descrita acima envolve compartilhamento de recursos e acesso a regiões críticas, tais como: alocação e liberação de objetos de recursos, alocação e liberação de worker threads, solicitação de OIDs nas tabelas de OIDs dos objetos remotos e locais, comunicação com o Ensemble para obtenção de novos OIDs. Para controlar o uso a estes recursos são utilizados monitores no sistema. Os monitores são procedimentos que impedem que dois componentes acessem ao mesmo tempo um recurso compartilhado. Quando um componente desejar acessar um dos recursos citados acima, ele solicitará este acesso ao monitor, o qual garantirá a exclusão mútua a esse recurso.

3.2.3 Modelos RPOO

Neste experimento, a notação RPOO foi utilizada para modelar o protocolo de comunicação entre objetos do sistema *DistBeta*. Nesta seção apresentamos apenas o diagrama de classes e o detalhamento de uma das classes do modelo RPOO. O modelo completo do Sistema *DistBeta* encontra-se no Apêndice B.

Diagrama de Classes

Na Figura 3.19 é apresentado o diagrama de classes do sistema *DistBeta*. O modelo possui oito classes: *ENSEMBLE*, *SHELL*, *REDE*, *THREADUSER*, *THREADLISTEN*, *THREADWORK*, *THREAD* e *OBJETO*. A relação entre as classes *REDE* e *ENSEMBLE* indica que para cada instância da classe *REDE* deverá existir pelos menos duas instâncias da classe *ENSEMBLE*.

As classes *THREADUSER, THREADLISTEN* e *THREADWORK*, ajudam a descrever a classe *THREAD* utilizando o mecanismo de agregação. A cardinalidade declarada pela associação destas classes indicam que pelo menos um *user thread* deve estar disponível no *shell* para iniciar a comunicação; deve existir apenas um *listener thread* por *Shell*; pode ser que nenhum *worker thread* seja instanciado no sistema, caso não seja solicitado nenhum serviço ao determinado *Shell*, ou pode ser que vários sejam instanciados.

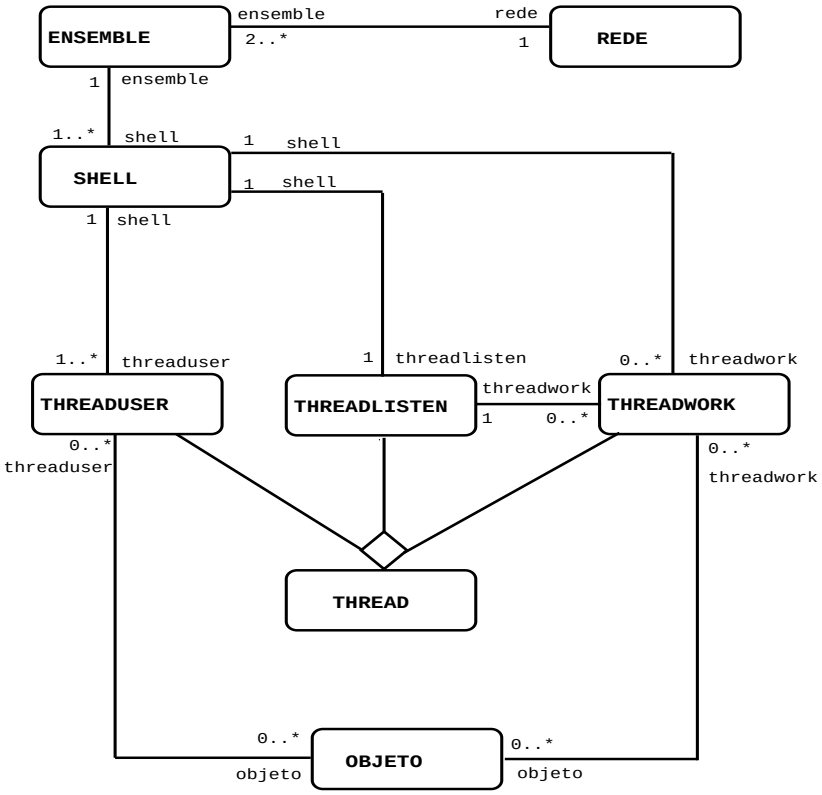


Figura 3.19: Diagrama de Classes

Classe OBJETO

Na Figura 3.20 é apresentado o corpo da classe *OBJETO*. O modelo consiste de três lugares: Executando, StandBy e Solicitando, representando que o objeto está executando, em estado de espera ou solicitando. O corpo da classe consiste de quatro transições: Executa_tarefa, Solicita_tarefa, Encerra_tarefa e Encerra_Solicitação. Inicialmente, existe uma ficha **e** no

lugar *StandBy* representando a marcação inicial da rede. A transição *Solicita_tarefa* estará habilitada para todos os objetos instanciados no sistema e será a única habilitada em todo o modelo quando este for iniciado. Quando a transição *Solicita_tarefa* disparar, uma ficha será removida do lugar *StandBy* e adicionada no lugar *Solicitando*, indicando que o objeto está solicitando uma informação de outro objeto remoto ao objeto *user thread*. Ao disparar esta transição, a mensagem *thuser!solicitar_obj* é enviada ao *user thread* ligado ao objeto, o qual é responsável pela execução da tarefa solicitada ao objeto. A partir do estado Solicitando, a transição *Encerra_Solicitacao* poderá ocorrer. A ocorrência desta transição modela o recebimento da mensagem *thuser?liberar* de um *user thread*. Após este recebimento o objeto voltará para o estado *standBy*.

OBJETO

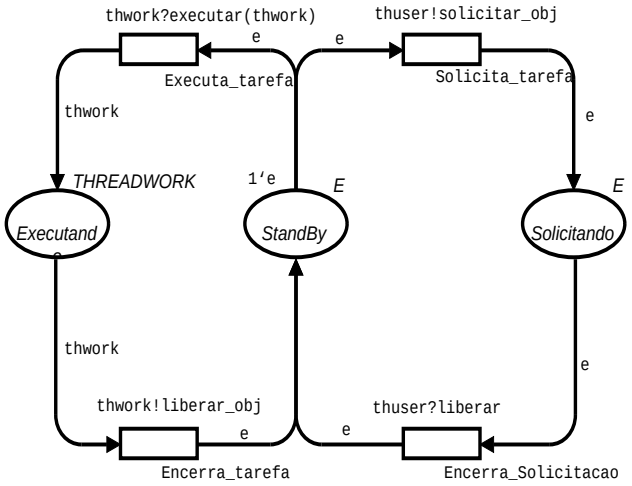


Figura 3.20: Classe OBJETO

A transição *Executa_tarefa* modela a situação em que um objeto instanciado em um *host* recebe a mensagem *thwork?executar(thwork)* do *worker thread* solicitando a execução de uma tarefa. O disparo da transição *Encerra_tarefa* faz com que a mensagem *thwork!liberar_obj* seja enviada para o *worker thread* que fez a solicitação, o qual estará aguardando do objeto o retorno de sua solicitação.

3.2.4 Análise do Experimento

Validação do Modelo RPOO

Para validar o modelo também utilizamos a ferramenta de simulação RPOO, ou seja, o Design/CPN para simular as redes de Petri, onde efetuamos o papel de mediador com o simulador, e o SSO para simular o sistema de objetos. Para efetuar a validação do modelo foram considerados diversos cenários. A Figura 3.18 mostra um dos cenários utilizados. Este cenário consiste em uma rede, dois *ensemble*, dois *shells*, dois *users threads*, dois *listeners threads*, um *worker thread* e dois objetos. Os resultados da simulação do sistema *DistBeta*, foram analisados a partir dos relatórios gerados pela ferramenta Design/CPN após a simulação de cada objeto do sistema. A ferramenta SSO executou as simulações do sistema de objetos através do processamento das inscrições de interação associadas às transições dos corpos das classes que foram disparadas durante a simulação das redes de Petri, gerando as mensagens para os respectivos objetos do modelo.

Na Figura 3.21 apresentamos o MSC gerado a partir de uma das simulações efetuadas. O MSC possui doze colunas: duas colunas *Objeto1* e *Objeto2*, representando os objetos *01 e 02*; duas colunas *Thuser1* e *Thuser2*, representando os objetos *user thread 1* e *user thread 2*; duas colunas *Shell1* e *Shell2*, representando os objetos *shell 1* e *shell 2*; duas colunas *Thlisten1* e *Thlisten2*, representando os objetos *listener thread 1* e *2*; uma coluna *Thwork1*, representando o objeto *worker thread 1*; duas colunas *Ensemble1* e *Ensemble2*, representando os objetos *ensemble 1* e *2*; e uma coluna *Rede*, representando o objeto *rede*.

A inscrição de interação *obj1:thuser1!solicitar_obj* (coluna *Objeto1*), indica que o *objeto 1* está enviando a mensagem *solicitar_obj* de maneira síncrona para o objeto *user thread* (1.etapa da comunicação entre objetos). A inscrição *thuser1:obj1?solicitar_obj* indica que o objeto *user thread* recebeu (executou uma ação ■) a mensagem *solicitar_obj* enviada pelo *objeto 1*. Como podemos observar na coluna *Thuser1* do MSC, quando o objeto *user thread 1* executa o processamento desta mensagem, ele envia a mensagem *thuser1:shell1!in_monitor(recurso)* para o *shell 1* solicitando o acesso ao recurso monitor. Ao receber a mensagem *thuser1:shell1?out_monitor* do objeto *shell 1*, o objeto *user thread 1* vai serializar a mensagem (■ *thuser1:thuser1!serializar @ thuser1:thuser1?serializar, thuser1:thuser1!end_serializacao @ thuser1:thuser1?end_serializacao*) antes de repassá-la ao

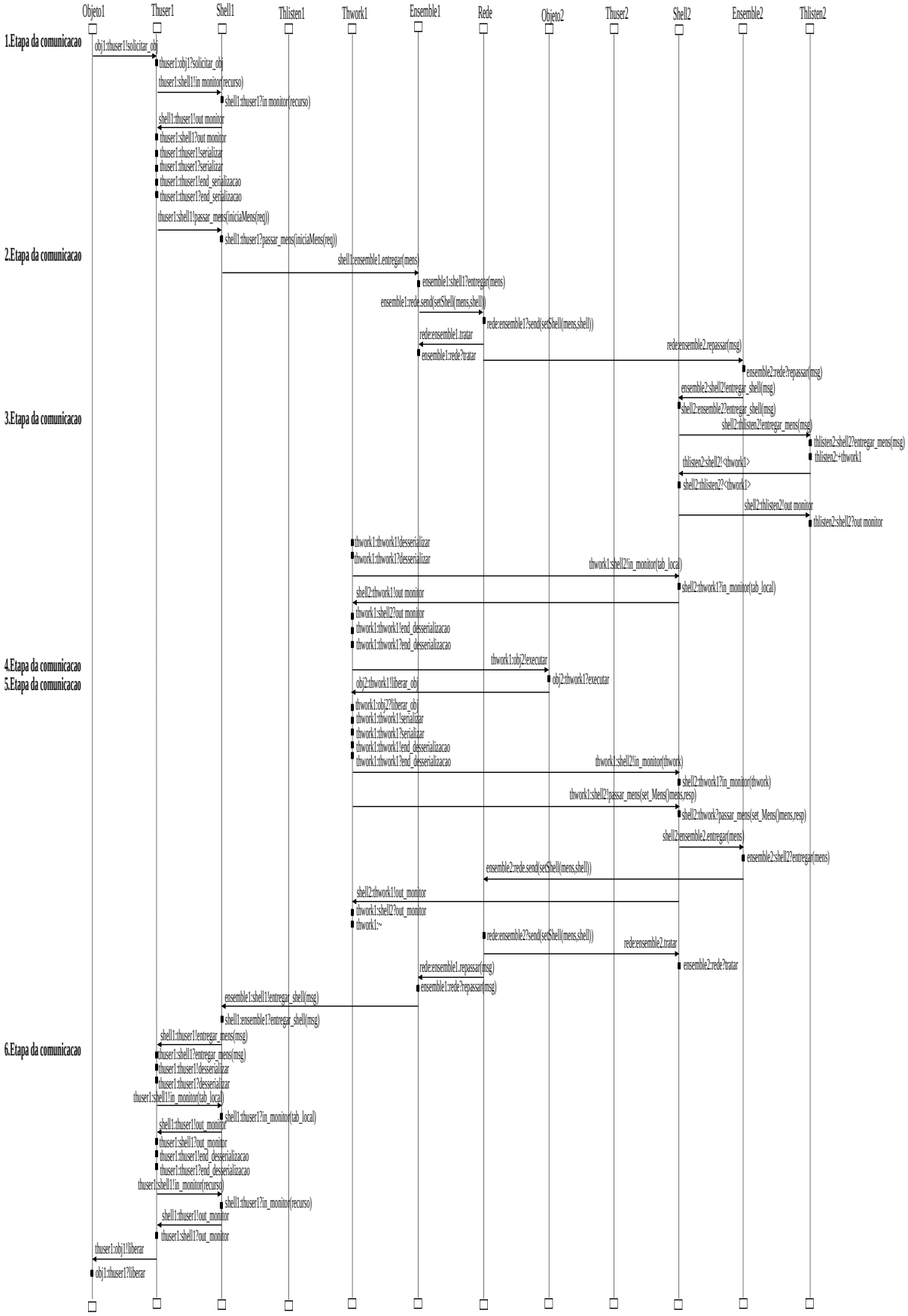


Figura 3.21: Diagrama de Sequência de Mensagens

objeto *shell 1*, através da mensagem *thuser1:shell1!passar_mens(iniciaMens(req))*(2.etapa da comunicação entre objetos).

A inscrição de interação ■ *thlisten2:+thwork1* na coluna *Thlisten2* indica que o objeto *listener thread 2* executou uma ação de criação do objeto *worker thread 1*, ou seja, o objeto *worker thread 1* foi criado pelo objeto *listener thread 2*.

O resultado obtido com a simulação do modelo *DistBeta* utilizando o SSO e a ferramenta Design/CPN como o simulador de redes de Petri na simulação de cada classe isoladamente, nos dá confiança de que o modelo está correto.

Impressões Sobre a Solução Obtida

Inicialmente, o modelador dedicou-se ao aprendizado do formalismo RPOO. Apesar de não ter nenhuma experiência com a notação, o mesmo não encontrou dificuldades com o aprendizado de RPOO. Como citado anteriormente o modelador possuía conhecimentos em CP-Net e OO. Assim, ele encontrou muita facilidade para trabalhar com a linguagem, pois a integração dos conceitos foi efetuada de maneira que nenhuma das características foram alteradas. Os produtos de modelagem foram desenvolvidos em um período de dois meses, com aproximadamente 05 horas de trabalho diário.

Dentre as facilidades encontradas pelo modelador, está a modelagem da comunicação entre as partes do sistema. Todas as trocas de mensagens entre os objetos foram efetuadas através das inscrições de interação.

A única dificuldade encontrada foi a falta de uma ferramenta mais completa de apoio computacional para análise e verificação dos modelos desenvolvidos.

Analisando os produtos de modelagem obtidos identificamos claramente todos os componentes do sistema, bem como os seus relacionamentos. Além disso, identificamos facilmente o comportamento dos componentes e as trocas de mensagens efetuadas entre eles.

Contribuições do Experimento

Este experimento consolidou os resultados do primeiro experimento quanto a metodologia utilizada para a modelagem de sistemas utilizando a notação RPOO, bem como ao uso da ferramenta SSO para efetuar a simulação do sistema de objetos.

3.3 Protocolo Open Shortest Path First – (OSPF)

3.3.1 Motivação e Objetivos

O *Open Shortest Path First* (OSPF) é o protocolo utilizado pela Internet para efetuar o roteamento de pacotes, sendo o mesmo utilizado pela rede da Universidade Federal da Paraíba.

O protocolo *OSPF* foi o terceiro sistema modelado em RPOO. Um dos fatores que contribuíram para esta escolha foi a facilidade de interação com o professor do Departamento de Sistemas e Computação (DSC) responsável pela rede desta Universidade, que possibilitou um completo entendimento do funcionamento do protocolo. Além disso, iniciou-se um projeto usando HCPN para avaliação de desempenho do sistema de roteamento deste departamento.

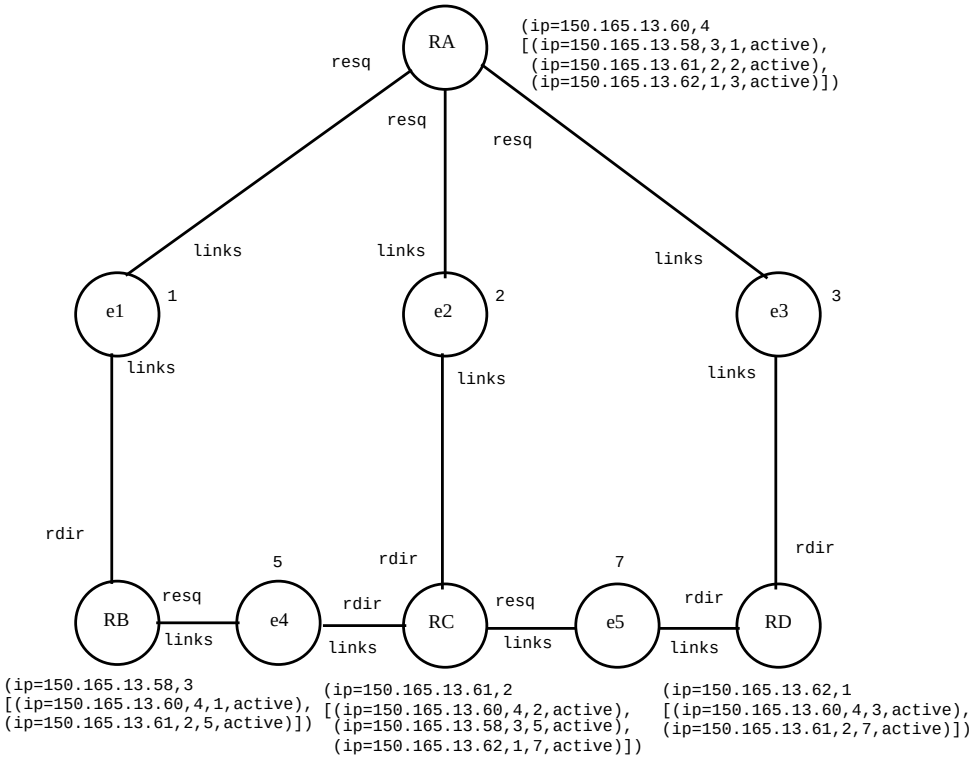


Figura 3.22: Configuração Local do Protocolo

A Figura 3.22 mostra a configuração do protocolo *OSPF* utilizada pelo DSC. Esta configuração consiste em quatro roteadores (*RA*, *RB*, *RC* e *RD*) e cinco enlaces (*e1*, *e2*, *e3*, *e4* e *e5*). As inscrições associadas aos nós *RA*, *RB*, *RC* e *RD* representam a configuração dos roteadores, as inscrições associadas a *e1*, *e2*, *e3*, *e4* e *e5* correspondem aos custos dos en-

laces. Embora o protocolo *OSPF* apresente características de distribuição e concorrência, quando estudamos o funcionamento da configuração local, identificamos que a configuração utilizada era muito restrita. Muitos dos aspectos do protocolo não foram contemplados, como por exemplo, a topologia de interconexão dinâmica. A topologia do protocolo é denifida como sendo estática. Assim, desenvolvemos este experimento com o intuito de obervar como seria usar RPOO na modelagem de um sistema, que não apresenta características desejáveis da notação RPOO.

Um dos objetivos deste terceiro experimento foi efetuar um estudo comparativo de modelagem do referido protocolo utilizando duas extensões de redes de Petri, já que o experimento foi desenvolvido baseado em um sistema que não atende especificamente aos propósitos de RPOO. Neste sentido, utilizamos os resultados obtidos por uma outra equipe que desenvolveu em paralelo um experimento de modelagem do mesmo problema utilizando redes de Petri coloridas hierárquicas. Esta equipe foi constituída por uma aluna do curso de mestrado do Departamento de Sistemas e Computação desta Universidade. A existência de dois modelos de um único sistema, desenvolvido usando duas extensões diferentes de redes de Petri, nos permite efetuar uma comparação entre os produtos de modelagem obtidos. Como os outros experimentos anteriores também possuem um produto de modelagem usando HCPN, optamos por fazer um estudo comparativo de modelagem para ambos os experimentos. Objetivamos ainda com este experimento identificar as dificuldades e facilidades encontradas pelos modeladores durante o desenvolvimento dos modelos utilizando ambas as abordagens.

3.3.2 O OSPF

O protocolo *Open Shortest Path First* (*OSPF*) é um protocolo de roteamento especialmente projetado para o ambiente Internet TCP/IP, o qual roteia pacotes IP baseando-se apenas no endereço IP de destino [Rfc; Tan97]. O *OSPF* foi desenvolvido por um grupo de trabalho da Internet Engineering Task Force, com o propósito de atender às exigências de roteamento de grandes redes, que é responder rapidamente às mudanças ocorridas na topologia da rede, tornando-se um protocolo padrão desde 1990.

O protocolo *OSPF* é baseado no algoritmo de estado de enlace, que possibilita ao roteador ter uma visão geral da topologia da rede. Podemos entender um enlace como um canal de comunicação entre dois elementos da rede. O estado deste enlace é a relação dele com

os roteadores da rede. Em um protocolo de roteamento por estado de enlace, cada roteador mantém uma base de dados individual descrevendo a topologia da rede.

As trocas de informações entre os roteadores são feitas através de pacotes de anúncio de estado de enlace (LSA's). Cada roteador recebe LSA's contendo os dados relevantes sobre os roteadores da rede para criar a sua base de dados. Depois de completar a sua base de dados, é calculado então, o melhor caminho para chegar ao roteador de destino através do algoritmo de menor caminho – *Shortest Path Algorithm* [Dij; THC90] definido por algoritmo de Dijkstra. O Algoritmo de Dijkstra é um algoritmo que calcula o caminho de custo mínimo entre vértices de um grafo. Escolhido um vértice como raiz da busca, este algoritmo calcula o custo mínimo deste vértice para todos os demais vértices do grafo. O roteador gera a sua tabela de roteamento a partir deste grafo. Todos os roteadores executam o mesmo algoritmo em paralelo.

Características do protocolo OSPF

Devido à grande experiência com outros protocolos de roteamento, o grupo que projetou o protocolo *OSPF* o definiu com as seguintes características:

- Protocolo de Gateway Interno (IGP) – o *OSPF* é classificado como IGP, por prover o roteamento de pacotes dentro de um Sistema Autonômo (conjunto de redes interligadas sob o controle de uma mesma autoridade administrativa);
- Especificação de domínio público – protocolo aberto;
- Algoritmo dinâmico – adapta-se de forma rápida e automática a alterações de topologia. No momento em que ocorre um problema em um enlace, automaticamente, o protocolo de roteamento ajusta a tabela de rotas para que a entrega das informações possa ser feita por outro caminho;
- Utiliza o algoritmo de roteamento de estado de enlace:
  - Melhor convergência do que algoritmos de roteamento de vetor distância[3];

[3]Os protocolos baseados no algoritmo de vetor distância partem do princípio de que cada roteador do Sistema Autonômo (AS) deve conter uma tabela informando todas as possíveis rotas dentro deste AS. A partir desta tabela o algoritmo escolhe a melhor rota e o enlace que deve ser utilizado. Estas rotas formam uma tabela.

  - Após o cálculo da tabela de roteamento as atualizações são transmitidas apenas nos casos de alterações nas rotas;

- Balanceamento de Carga – quando existem várias rotas de custos iguais para um destino, o tráfego é distribuído igualmente entre elas.

O protocolo *OSPF* é compatível com três tipos de conexões e redes : linhas ponto a ponto; redes de multiacesso com difusão (por exemplo, a maioria das LANs); redes de multiacesso sem difusão (por exemplo, a maioria das WANs de comutação de pacotes).

Algoritmo de Roteamento

O algoritmo de roteamento por estado de enlace é simples e pode ser estabelecido em cinco partes. Cada roteador deve fazer o seguinte:

1. descobrir os roteadores vizinhos e aprender seus endereços de rede – quando um roteador é iniciado, ele envia um pacote Hello em cada interface de saída para descobrir os seus vizinhos;

2. medir o custo para cada um de seus vizinhos;

3. construir pacotes de anúncio de estado de enlace (LSA/Database Description), que contém informações sobre todos os vizinhos;

4. transmitir esse pacote a todos os outros roteadores da rede;

5. construir um grafo com a topologia da rede, a partir desse grafo o roteador calcula o melhor caminho para cada um dos outros roteadores – utiliza o algoritmo de Dijkstra (*Shortest Path Algorithm*) para calcular os caminhos mais curtos até os demais roteadores. Os menores caminhos são registrados na tabela de roteamento e o roteador volta a operação normal.

Os roteadores transmitem pacotes periodicamente informando a sua tabela, gerando um aumento de tráfego na rede.

Tipos de Mensagens

O protocolo *OSPF* possui os seguintes tipos de pacotes:

1. **Hello** – quando um roteador é iniciado, sua primeira tarefa é aprender quem são seus vizinhos. Isto é feito enviando pacotes Hello, transmitindo-os ao grupo formado por todos os outros roteadores. O roteador receptor deve enviar de volta uma resposta dizendo quem é, ou seja, seu endereço de rede. O pacote Hello em redes multi-acesso broadcast ou não-broadcast também é usado para eleger o roteador designado, que é responsável por gerar os LSA's. O roteador designado (RD) permite uma redução no tráfego da rede, reduzindo o tamanho da base de dados topologica. Quando as bases de dados de dois roteadores são sincronizadas, eles são ditos adjacentes. O roteador designado em redes multi-acesso é responsável por determinar que roteadores podem tornar-se adjacentes. Em redes multi-acesso não-broadcast cada roteador tem a lista de todos os seus roteadores vizinhos anexados a rede.

2. **Database Description** – anúncio do estado de enlace (LSA's) ao roteador designado, o qual repassará a todos os demais roteadores da rede.

3. **Link State Request** – são usados para requisitar os estados de enlace dos roteadores.

4. **Link State Update** – são utilizados para fornecer os custos do transmissor aos seus vizinhos.

5. **Link State Ack** – este pacote é usado para fazer a confirmação da atualização do estado de enlace, os quais são enviados em resposta aos pacotes *Link State Update*.

3.3.3 Modelos RPOO

Nesta seção apresentamos alguns resultados dos produtos de modelagem do protocolo *OSPF* utilizando a linguagem RPOO e a ferramenta Design/CPN para edição do modelo.

Durante o processo de modelagem foram feitas algumas considerações em relação ao funcionamento do protocolo, devido ao fato de estarmos modelando uma configuração específica do mesmo:

1. Todos os roteadores são iniciados com sucesso.

2. Um roteador somente poderá falhar depois que estiver com a tabela de roteamento calculada.

3. Qualquer roteador pode se recuperar de uma falha, tornando-se novamente ativo.

4. Alguns tipos de pacotes não foram considerados: *Link State Request, Link State Update e Link State Ack.*

5. Tratamos do funcionamento do protocolo em uma rede de multiacesso sem difusão (redes não broadcast). Neste tipo de rede não é formado adjacências entre dois roteadores e o roteador designado é somente responsável por gerar LSA's aos demais roteadores da rede.

Diagrama de Classes

Na Figura 3.23 é apresentado o diagrama de classes do modelo. Identificamos seis classes: *ROUTER, Designated Router, LSA, Routing, Failure* e *LINK*. A associação **are_linked** declara como os roteadores são relacionados ao enlace. Assim, dois roteadores podem ser relacionados por meio do objeto link. As inscrições usadas nas associações declaram a cardinalidade e os seletores. Consequentemente, cada enlace (link) deve conectar dois roteadores e cada roteador deve ser conectado por pelo menos um enlace. Optamos modelar a classe *LINK* porque os roteadores roteiam pacotes utilizando o enlace para transmití-los. Assim, o enlace é um componente ativo no roteamento de pacotes.

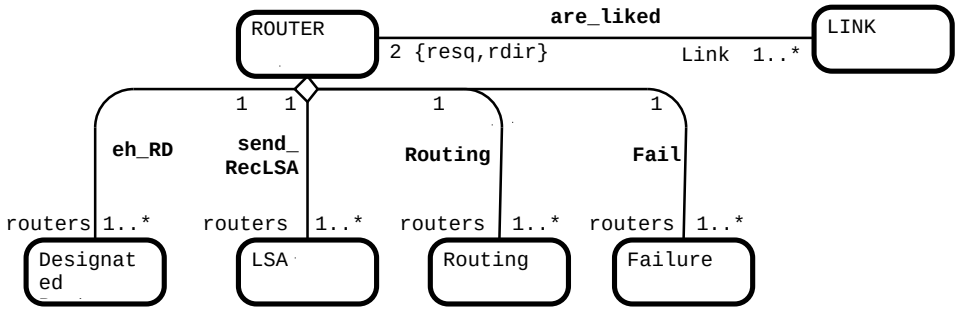


Figura 3.23: Diagrama de Classes

As classes *Designated Router, LSA, Routing* e *Failure*, ajudam a descrever a classe *ROUTER* utilizando o mecanismo de agregação, estando associadas a ela através dos seletores **eh_DR, Send_RecLSA, Routing e Fail**. Isto significa que para cada roteador ativo no modelo um roteador é roteador designado dele; um roteador irá enviar e receber LSA's; o roteador irá rotear pacotes de dados, bem como um roteador poderá falhar.

Classe Designated Router

Na Figura 3.24 é apresentado o corpo da classe *Designated Router*, a qual consiste em uma rede com três lugares: *Start*, *With Hello* e *Router With DR*, representando que o roteador está pronto para iniciar, recebeu o reconhecimento dos pacotes Hello, e elegeu o roteador designado, respectivamente.

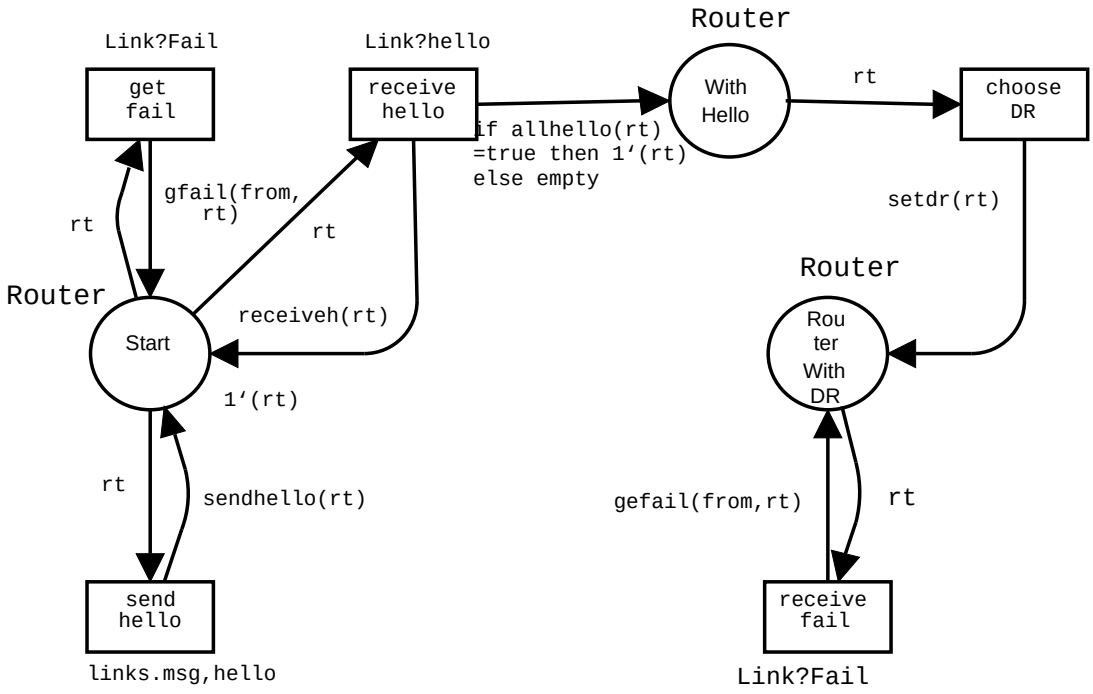


Figura 3.24: Classe Designated Router

A Classe *Designated Router* consiste de cinco transições: *send hello, get fail, receive hello, choose DR* e *receive fail*. A marcação da rede determina o estado inicial dos objetos instanciados daquela classe. Logo, um objeto roteador tem como estado inicial a marcação **rt** no lugar *Start*, significando que o roteador está pronto para trocar pacotes *hello*. Inicialmente, apenas as transições *send hello, receive hello* e *get fail* estão habilitadas. A ocorrência da

transição *send hello* modela a situação na qual o roteador envia a mensagem *links.msg,hello* a todos os seus vizinhos. Quando um roteador é iniciado ele pode tanto enviar uma mensagem *hello* para seus vizinhos como também receber uma mensagem *hello* de um dos seus vizinhos. No momento em que o roteador está enviando ou recebendo a mensagem *hello* de seus vizinhos, ele também pode receber uma mensagem de algum roteador que falhou. Isto é modelado pela inscrição de interação *Link?Fail* da transição *get fail*.

Quando o roteador enviar uma mensagem *hello* para os seus vizinhos ativos na rede, ele receberá de volta uma mensagem *hello* de todos os seus vizinhos (inscrição de interação *Link?hello*). A função *receiveh* é responsável por atualizar os parâmetros do roteador para indicar que todos os seus vizinhos enviaram o reconhecimento dos pacotes *hello*. A ocorrência da transição *receive hello* somente irá depositar uma ficha no lugar *With Hello*, quando a função *allhello* for satisfeita. O detalhamento destas funções encontra-se no nó de declaração do modelo. A partir do estado *With Hello*, o roteador estará apto a escolher o seu roteador designado. A escolha do roteador designado é representada no modelo pela *transição choose DR*, estando a função *setdr* responsável por realizar tal escolha. A escolha do roteador designado é feita selecionando o roteador de maior prioridade e maior endereço IP, o qual tem como funcionalidade repassar todos os pacotes recebidos de um roteador para os demais roteadores ativos na rede. Depois que o roteador conhecer o seu roteador designado, estado em que há uma ficha no lugar *Router With DR*, ele estará apto a receber a mensagem *Link?Fail* de qualquer outro roteador da rede (transição *receive fail*), e a transmitir ou receber pacotes *Description Database*, os quais contém o anúncio do estado de enlace do roteador (LSA's).

O modelo completo do protocolo *OSPF* encontra-se no Apêndice C.

3.3.4 Análise do Experimento

Validação do Modelo RPOO

Para efetuar a validação do modelo utilizamos a ferramenta de simulação RPOO. Foram considerados diversos cenários para efetuar a validação do modelo. A Figura 3.22 mostra uma das configurações utilizadas.

A Figura 3.25 descreve um dos MSCs gerados a partir das simulações efetuadas utili-

zando a ferramenta Design/CPN e a ferramenta SSO. O MSC possui nove colunas: quatro colunas (*RA, RB, RC* e *RD*), representando os objetos roteadores ativos no modelo, e as colunas restantes (*e1,e2,e3,e4* e *e5*) representando os objetos *links* conectando os roteadores.

A partir do MSC descrito na figura 3.25 é possível observar que as inscrições de interação *RA:e1.hello,RA:e2.hello* e *RA:e3.hello*, indicam que o objeto *RA* está enviando a mensagem *hello* de maneira assíncrona para os objetos *e1,e2* e *e3*. Quando um dos objetos *links* receber a mensagem *hello*, ele irá efetuar o tratamento desta mensagem. Os processos marcados ■ *e1:RA?hello* e *e1:e1!tratar_msg,m @ e1:e1?tratar_msg,m* indicam que o link *e1* recebeu com sucesso a mensagem *hello* e a tratou apropriadamente. Processamentos similares são executados pelos outros *links*. A inscrição de interação *RA:e1!DDatabase* indica que o objeto *RA* está enviando uma mensagem *Description Database* de maneira síncrona para o objeto *e1*.

As simulações executadas utilizando a ferramenta Design/CPN e a ferramenta SSO, aumentaram nossa confiança na corretude do modelo, pois os resultados obtidos após a simulação do modelo do protocolo *OSPF* foram compatíveis com o comportamento esperado.

Impressões Sobre a Solução Obtida

O diagrama de classes do modelo nos fornece uma visão geral do sistema, permitindo identificar claramente todos os componentes e seus relacionamentos. No detalhamento dos corpos das classes *Router, Designated Router, LSA, Routing, Failure* e *Link*, podemos identificar facilmente o comportamento dos objetos roteadores e as trocas de mensagens efetuadas entre eles para a troca de pacotes.

Tivemos uma grande facilidade em modelar as trocas de pacotes entre os objetos do sistema. Dentre as dificuldades encontradas está a falta de uma ferramenta mais completa de apoio computacional para análise e verificação dos modelos desenvolvidos.

Dentre as dificuldades encontradas pela modeladora que desenvolveu a modelagem do protocolo *OSPF* utilizando a linguagem HCPN está a modelagem da comunicação entre os objetos do sistema. A dificuldade maior foi a necessidade de implementação de diversas funções ML [D.U94] para fazer o controle da troca de pacotes. A grande desvantagem disso é que um grande esforço que poderia ter sido voltado para a modelagem da funcionalidade do protocolo acabou sendo voltado para a implementação das funções.

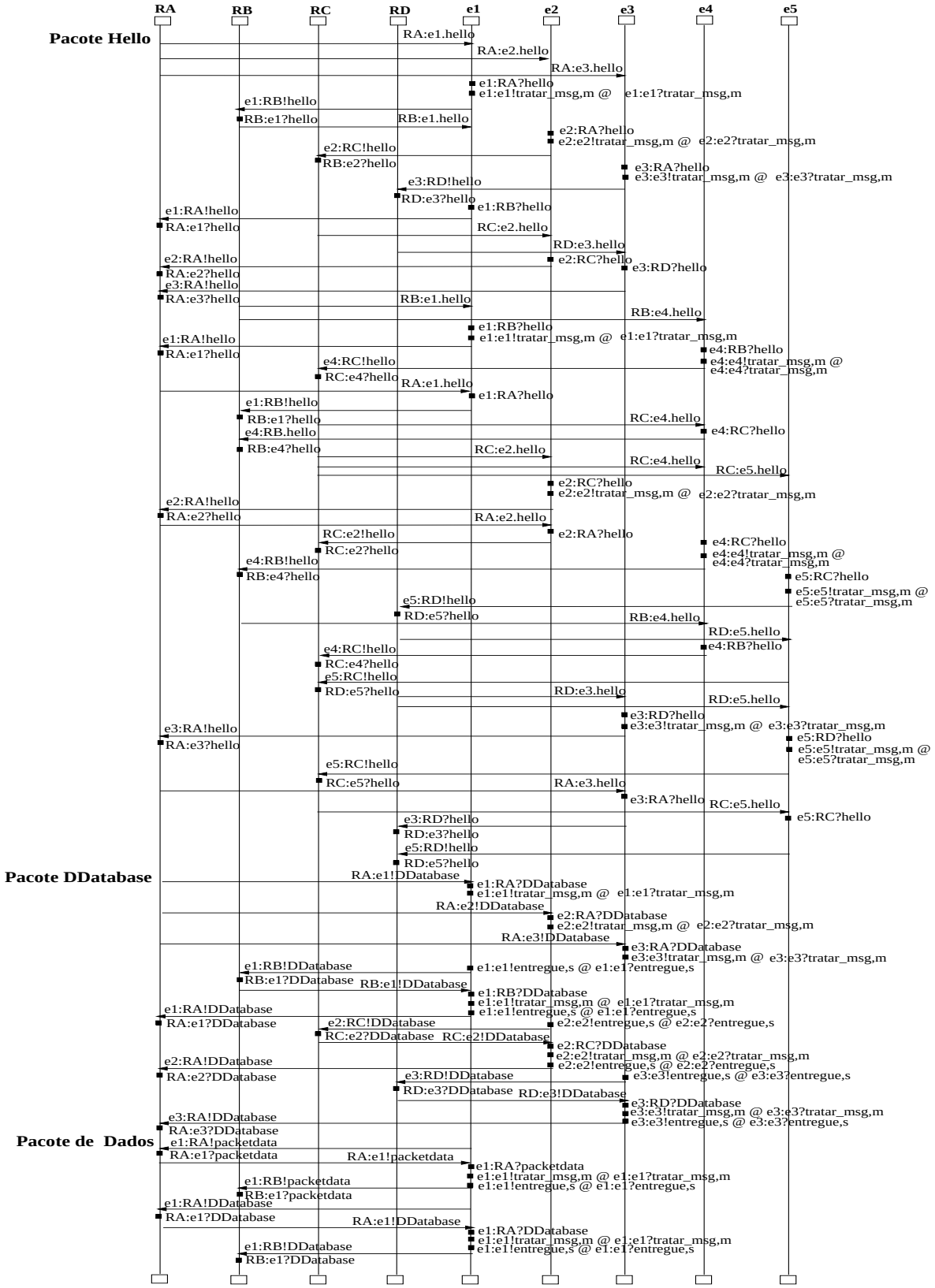


Figura 3.25: Diagrama de Sequência de Mensagens

Embora a linguagem RPOO tenha sido desenvolvida para a modelagem de sistemas distribuídos, caracterizados por topologias de interconexão dinâmica, neste experimento ela também se mostrou adequada para modelar todas as características de um sistema com topologia de interconexão estática.

Contribuições do Experimento

Para desenvolver o experimento de modelagem do protocolo *OSPF* utilizamos a mesma técnica adotada para decompor e estruturar os experimentos anteriores, ou seja, identificação dos componentes do sistema; identificação das classes do modelo e suas associações; elaboração do diagrama de classes; elaboração do diagrama de fluxo de mensagens; elaboração do diagrama de configuração inicial; e detalhamento do corpo das classes.

O estudo comparativo de modelagem entre os produtos obtidos utilizando a linguagem HCPN e a linguagem RPOO, encontra-se na Seção 4.2 do Capítulo 4.

Capítulo 4

Estudo Comparativo

Neste capítulo é apresentado um estudo comparativo de modelagem entre os modelos desenvolvidos utilizando a linguagem HCPN e RPOO para os experimentos apresentados no capítulo anterior. O objetivo deste estudo comparativo é avaliar o quão adequadas foram as duas abordagens em cada um dos experimentos escolhidos, a fim de identificar qual delas melhor representa o sistema modelado. Estudos comparativos têm demonstrado que não é fácil determinar um método para comparar linguagens de modelagem em relação a seus produtos [Bed95]. Um processo comparativo de modelagem pode ser influenciado por diversos fatores. Um importante fator é a adequabilidade do modelador a uma ou outra abordagem, bem como suas experiências passadas com outras linguagens. Ao se deparar com determinado artefato da linguagem, o modelador é influenciado pela prévia exposição a artefatos semelhantes de outras linguagens. De certa forma, isto induz o modelador a aceitar mais facilmente os mecanismos já dominados de modelagem e a rejeitar mecanismos diferentes, ainda que se usados corretamente sejam mais adequados.

Neste capítulo iremos dar enfase no detalhamento do experimento de modelagem do serviço *Bouncer*. A Seção 4.1 apresenta o serviço *Bouncer* em dois tópicos: i) detalhamento da modelagem usando HCPN; e, ii) impressões sobre a solução obtida, bem como a validação dos produtos de modelagem. Na Seção 4.2 é apresentado um estudo comparativo dos modelos obtidos em HCPN e em RPOO, para os três experimentos.

4.1 Modelos HCPN

Nesta seção apresentamos os resultados obtidos na modelagem do serviço *Bouncer* utilizando a linguagem HCPN e a ferramenta Design/CPN para edição, simulação e validação do modelo.

4.1.1 Página de Hierarquia

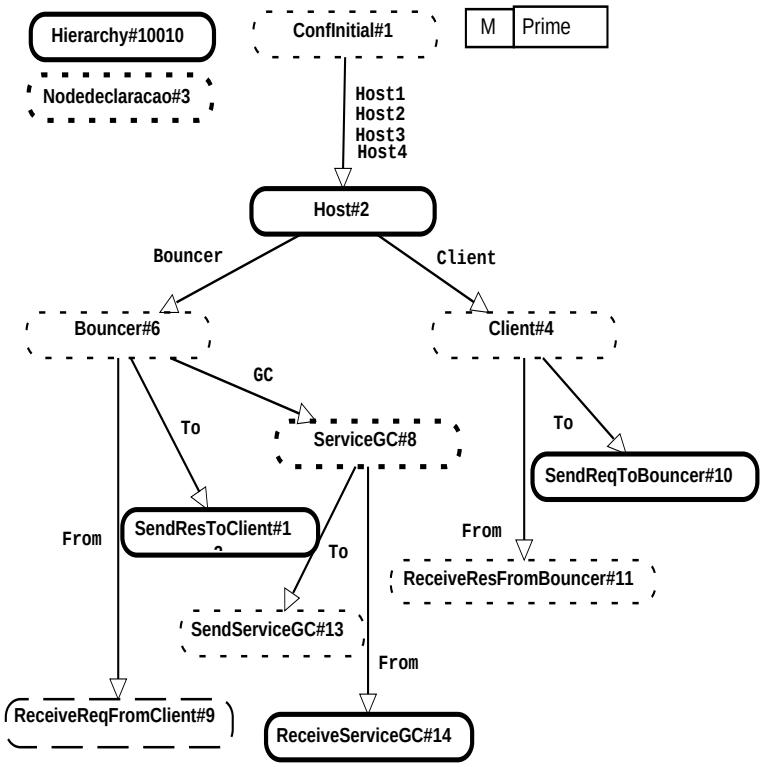


Figura 4.1: Página de *Hierarquia*

Na Figura 4.1 é apresentada a página de *hierarquia*. O modelo possui um total de 13 páginas. As páginas *Host*, *Bouncer*, *Client* e *Service GC* modelam os componentes *Host*, *Servidor Bouncer*, *Processo Cliente* e *Serviço de Comunicação em Grupo*, respectivamente. Essas páginas são abstrações desses componentes. As subpáginas *ReceiveReqFromClient*, *SendResToClient*, *ReceiveResFromBouncer*, *SendReqToBouncer*, *ReceiveServiceGC* e *SendServiceGC* detalham o funcionamento destes componentes.

As páginas *ReceiveReqFromClient* e *SendResToClient* modelam a comunicação do servidor Bouncer com o processo cliente. As páginas *ReceiveResFromBouncer* e *SendReqTo-*

Bouncer modelam a comunicação do processo cliente com o servidor Bouncer. Finalmente, as páginas *ReceiveServiceGC* e *SendServiceGC* modelam a comunicação entre os servidores bouncers ativos na rede local. As páginas que modelam a comunicação entre os componentes possuem comportamento semelhante. Assim, para evitar tornar o texto cansativo, apenas as páginas com bordas mais escuras na Figura 4.1, serão descritas.

A página *ConfInitial* é usada para iniciar as máquinas (*Hosts*), não representando um componente do sistema real, apenas um componente abstrato do modelo. As inscrições *Host1*, *Host2*, *Host3* e *Host4* do arco com origem na página *ConfInitial* indicam o número de instâncias do componente *Host* iniciados no modelo. Portanto, o modelo apresentado determina uma configuração inicial do sistema, consistindo em uma rede com quatro *hosts*.

4.1.2 Página Host

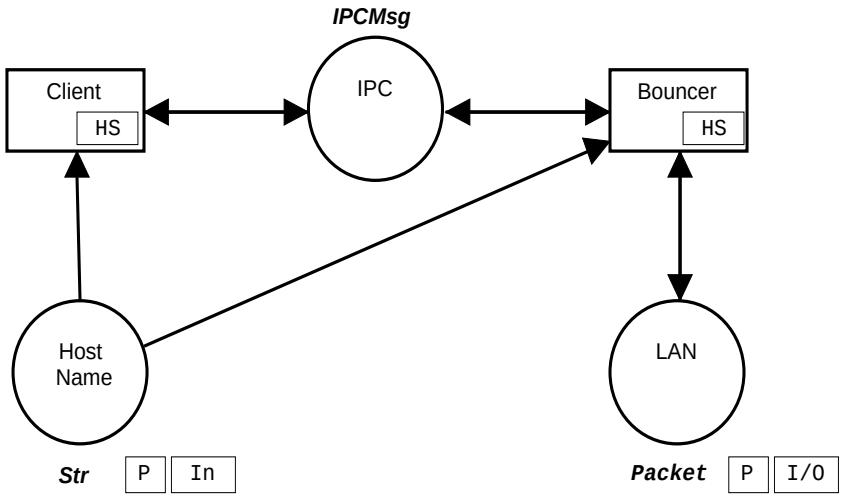


Figura 4.2: *Host*

Na Figura 4.2 é apresentada a página *Host*. Esta página consiste em uma rede com três lugares: *Host Name*, *IPC* e *LAN* e duas transições de substituição: *Client* e *Bouncer*. A ficha do lugar *Host Name* fornece o nome da máquina, que identifica a máquina onde o processo cliente e o servidor Bouncer estarão executando. Os lugares *IPC* e *LAN* funcionam como um buffer de mensagens. O processo cliente comunica-se com o servidor Bouncer de sua máquina através do lugar *IPC*—comunicação inter processos. O servidor Bouncer comunica-se com outros servidores da rede local através do lugar *LAN*. As transições de

substituição *Client* e *Bouncer* representam uma abstração do processo cliente e do servidor Bouncer instanciados no *Host*.

4.1.3 Página SendReqToBouncer

Na Figura 4.3 é apresentada a página *SendReqToBouncer*. Esta rede modela as requisições de serviços que o processo cliente pode fazer ao servidor Bouncer de sua máquina. Estes serviços são:

1. Requisitar ou iniciar um servidor Bouncer – caso não exista um servidor ativo na máquina, o processo cliente irá requisitar a sua iniciação, transição *request bouncer*.
2. Requisitar uma licença, transição *request_license*.
3. Liberar uma licença – antes de finalizar a sua execução, o processo cliente irá requisitar a liberação da licença que ele detém, transição *request release*.

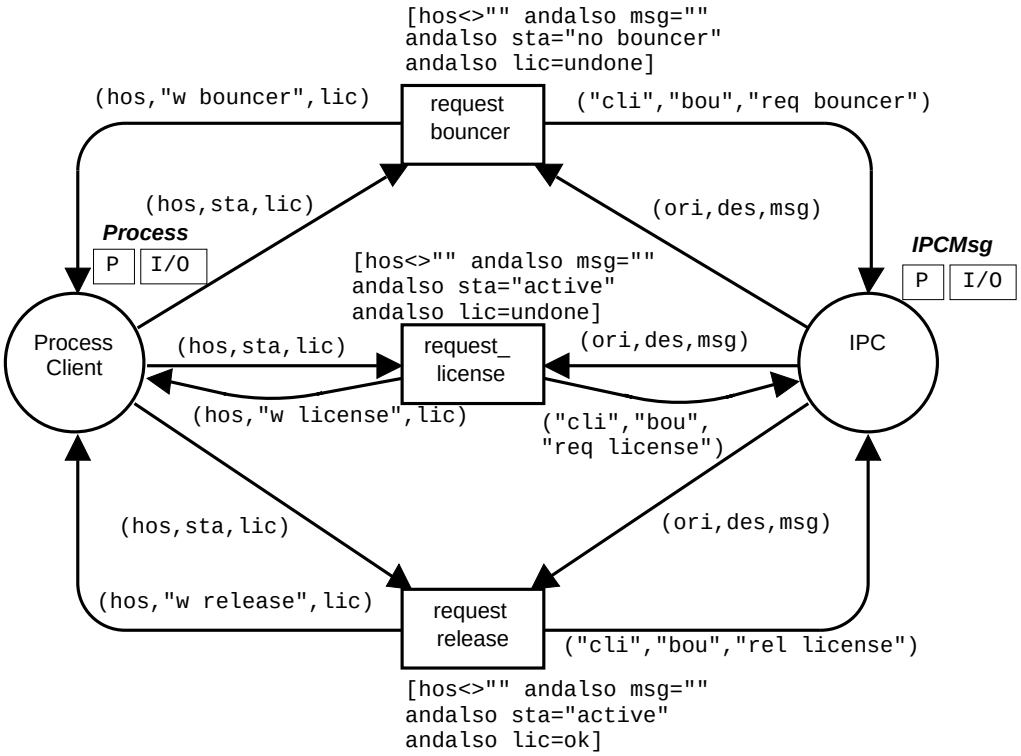


Figura 4.3: Comunicação do Processo Cliente com o Servidor Bouncer

Todas as mensagens de requisições de serviços do processo cliente são depositadas no lugar *IPC*. Quando o processo cliente enviar uma requisição de serviço ao servidor Bouncer,

seu status será alterado e ele ficará aguardando o serviço requisitado. O processo cliente estará apto a solicitar ou liberar uma licença ao servidor Bouncer caso o servidor esteja ativo, como podemos observar na guarda das transições *sta=“active”*.

4.1.4 Página SendResToClient

Na Figura 4.4 é apresentada página *SendResToClient*. Esta rede modela as respostas do servidor Bouncer às requisições de serviços feitas pelo processo cliente. As transições *bouncer requested, lic requested* e *released requested* modelam a situação em que o servidor Bouncer responde a uma requisição de um bouncer, uma requisição de licença, uma liberação de licença, respectivamente. A função *SendRes* é responsável por esta resposta. As guardas destas transições garantem que as respostas das requisições vão ser enviadas aos processos clientes que as requisitaram (*hasMsgToCB(nms)*); que o *buffer* de mensagens está vazio (*ipcIsEmpty(msg)*, pois as mensagens são recebidas em ordem; e, que para o servidor bouncer responder a uma requisição de serviço ele tem que estar ativo (*emptyB(bid)=false*), respectivamente.

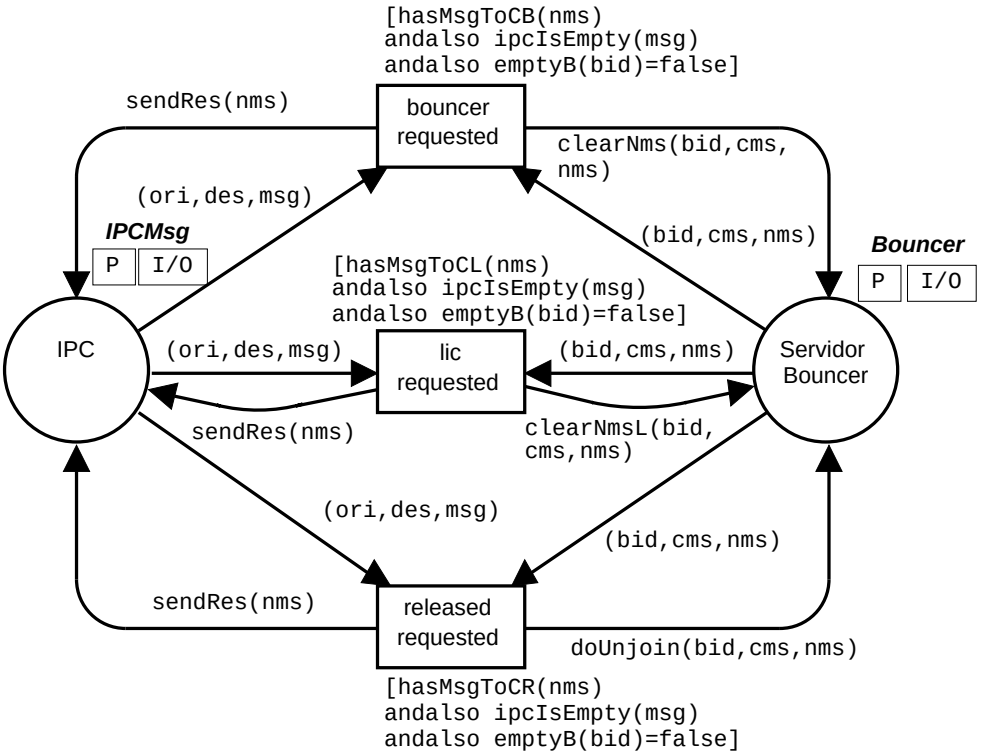


Figura 4.4: Comunicação do Servidor Bouncer com o Processo Cliente

Quando a transição *released requested* ocorrer, o bouncer verificará a existência de outros processos clientes utilizando o seu serviço. Caso não exista, será enviado uma mensagem de *Unjoin*—saída do grupo bouncer aos demais servidores da rede, através da função *doUnjoin*.

4.1.5 Página ReceiveServiceGC

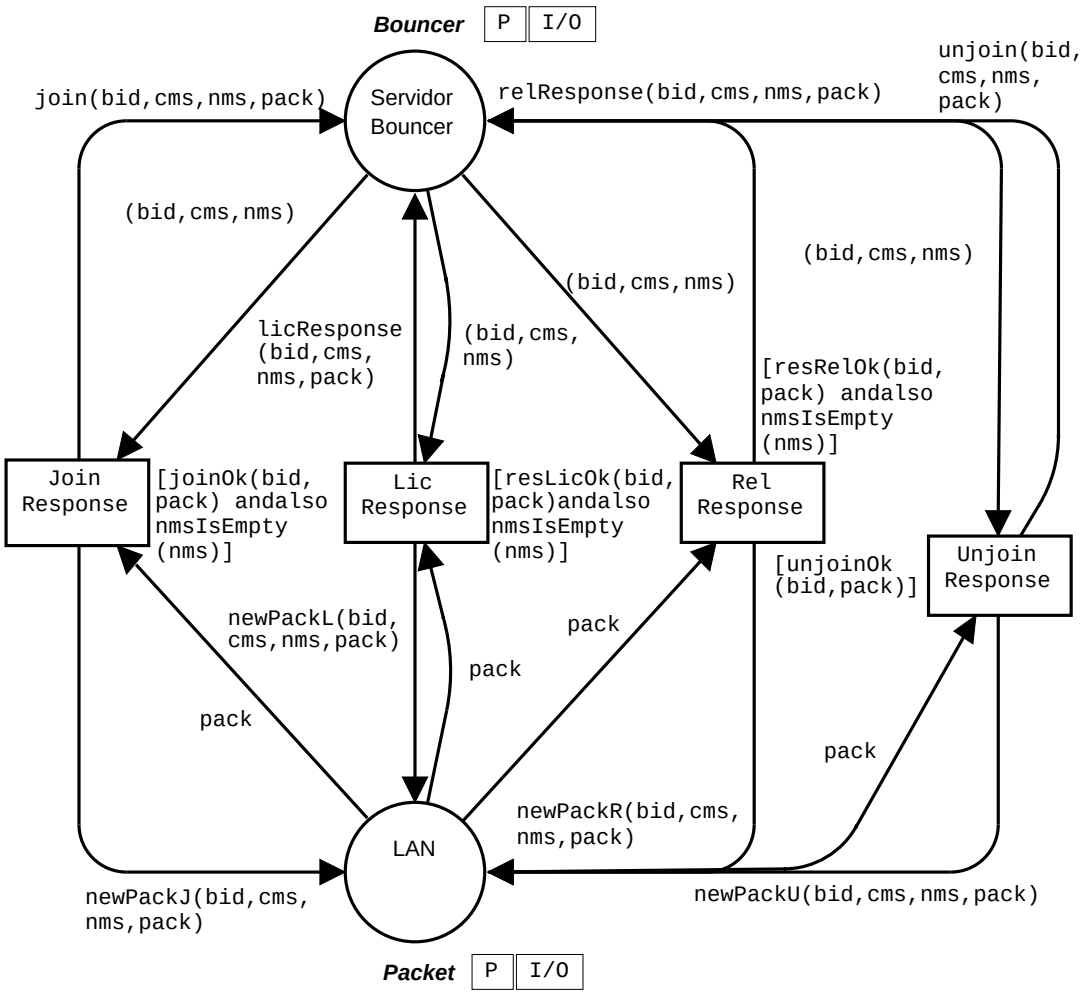


Figura 4.5: Comunicação entre os Servidores Bouncers—Grupo Bouncer

Na Figura 4.5 é apresentada a página *ReceiveServiceGC*. Quando o servidor Bouncer receber uma requisição do processo cliente, ele irá repassá-la para os demais servidores da rede local—grupo bouncer, incluindo ele mesmo. O conjunto de cores do lugar *LAN*, possui um campo (lista com todos os servidores Bouncers ativos na rede) que controla o recebimento das mensagens por todos os servidores. Se a requisição recebida for um ingresso no grupo bouncer transição *Join Response*, o servidor Bouncer será adicionado ao grupo.

Caso o servidor seja o primeiro elemento a ingressar no grupo, ele será o líder do grupo bouncer— procedimento modelado pela função *join*. Quando um servidor Bouncer terminar a sua execução, será enviado aos demais servidores uma requisição de saída do grupo bouncer, transição *Unjoin Response*. Ao receber esta requisição, cada servidor removerá do grupo bouncer o membro que efetuou esta solicitação, a função *unjoin* é responsável por esta exclusão. As novas requisições de serviços que forem enviadas ao grupo bouncer não serão mais recebidas pelo membro excluído. Ao responder uma requisição de licença, ocorrência da transição *Lic Response*, o servidor Bouncer verificará se o número de licença disponível na base de dados local é igual a zero. Se for, o pedido de licença será negado, e a resposta ao processo cliente será *lic denied*. Existindo licenças disponíveis a resposta enviada ao processo cliente será *lic ok* e o servidor Bouncer irá decrementar a sua tabela, comportamento modelado pela função *licResponse*. Se a requisição for uma liberação de licença, transição *Rel Response*, o servidor Bouncer irá incrementar a sua tabela local e responderá ao processo client com a mensagem *rel license*, representado no modelo pela função *relResponse*.

O modelo completo do serviço *Bouncer* utilizando a linguagem HCPN encontra-se no Apêndice A.

4.1.6 Impressões Sobre a Solução Obtida

A linguagem HCPN não fornece uma metodologia para estruturar e decompor os modelos. O que existe é uma certa abordagem padronizada, que corresponde à prática de modelagem de usários HCPN. Conseqüentemente, um mesmo sistema pode ser modelado adotando abstrações bastante diferentes. No início deste trabalho tentamos mais de uma vez modelar o *Bouncer*. Por exemplo, a página de hierarquia produzida no início do processo de modelagem era bastante diferente—ver Figura 4.6. Não se trata apenas de diferenças em detalhamento, mas na própria identificação dos componentes e abstrações. Esta solução não permite a identificação dos componentes do modelo. Estes componentes foram modelados em uma mesma rede. A página *host* contém uma abstração de todos os componentes do modelo, o que não é possível identificar observando a figura. Os serviços oferecidos pelos componentes é que foram detalhados em uma rede separada. Há dois motivos para isto. Em primeiro lugar, o fato de que HCPN não é vinculada a nenhuma metodologia de modelagem específica, proporcionando liberdade total ao modelador. O segundo motivo é que, devido

à falta de experiência de modelagem em HCPN, durante o desenvolvimento do modelo não adotamos nenhuma estratégia de modelagem. Em geral, essas estratégias são identificadas à medida que adquirimos experiência de modelagem. Por esta razão o modelo sofreu grandes modificações a partir do momento que passamos a ser supervisionado/acessorado por um modelador mais experiente em HCPN.

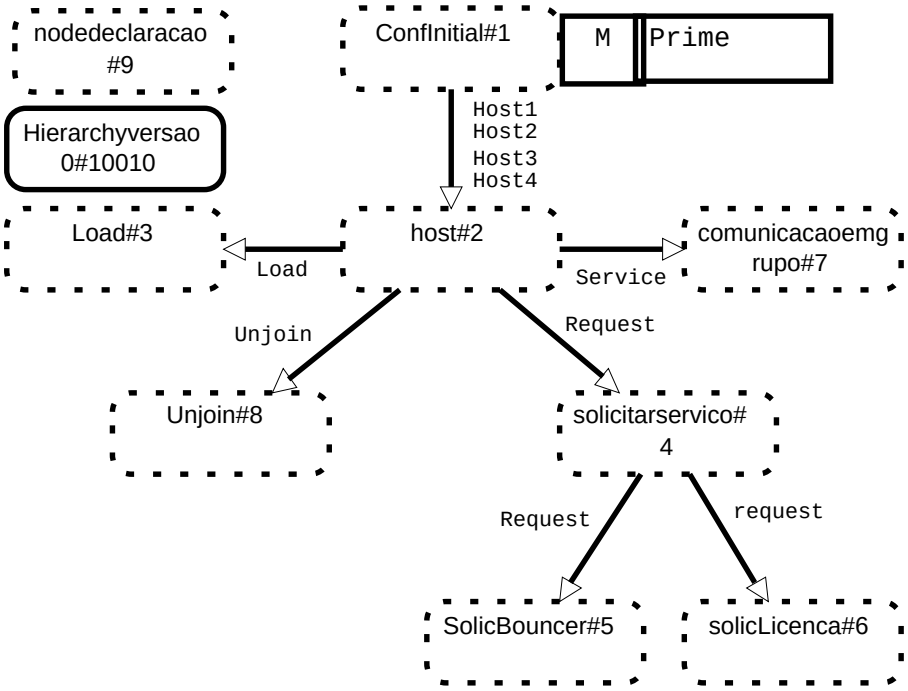


Figura 4.6: Página de *Hierarquia*– Outra Solução HCPN

Inicialmente, modelamos o funcionamento geral do serviço *Bouncer*. A partir da visão abstrata do modelo, foram identificadas partes que poderiam ser mais detalhadas. O serviço *Bouncer* foi modelado portanto de uma maneira *top-down*, partindo de um nível mais abstrato do mesmo até chegar nos detalhes de seu funcionamento – detalhamento das transições de substituição. A abstração oferecida por HCPN não permitiu a modelagem direta da criação e destruição de componentes. Para prover este comportamento foi necessário compartilhar estados dos componentes do sistema, efetuando o controle através da codificação das funções e dos tipos de dados.

A principal dificuldade enfrentada durante o desenvolvimento dos produtos de modelagem do serviço *Bouncer* foi a modelagem da comunicação entre os componentes. Como HCPN não oferece mecanismos para efetuar a comunicação direta entre os componentes, para prover este mecanismo optamos pela modelagem da comunicação utilizando fun-

ções ML no nó de declarações do modelo. Isto impôs a implementação de diversas funções ML para fazer o controle das requisições de serviços. Por exemplo, a função *licResponse* usada para responder a uma requisição de licença—Figura 4.7.

```
fun licResponse (bid,cms,nms,((ori,des,content),checklist)) =
let
 val hos=getHost(bid)
 val sta=getStatus(bid)
 val lid=getLider(bid)
 val gro=getGroup(bid)
 val pkdescr=getPkDescricao(content)
 val pklic=getPkLicenca(content)
 val pkgro=getPkGroup(content)
 val pklid=getPkLider(content)
 val chsize=getSize(checklist)
 val gr1=incGr(ori,gro)
 val gr2=decGr(ori,gro)
 val num=getNumLics(bid)
in
 if num=MAXLIC andalso sta="active" andalso ori=hos
    then ((hos,sta,lid,gro),cms,("bou","cli",("lic den","",[])) )

 else if num<MAXLIC andalso sta="active" andalso   ori=hos
    then  ((hos,sta,lid,gr1),cms,("bou","cli",("lic ok","",[])) )

    else if sta="active" andalso ori<>hos
    then ((hos,sta,lid,gr1),cms,nms)

 else (bid,cms,nms)

end;
```

Figura 4.7: Função *licResponse*

Dentre as facilidades encontradas está o fato das redes de Petri coloridas hierárquicas terem uma ferramenta de suporte computacional para simulação e análise dos modelos. Todos os recursos necessários para a modelagem e validação do experimento foram oferecidos pela ferramenta Design/CPN.

A validação do modelo foi efetuada usando a ferramenta de espaço de estados do simulador Design/CPN. Foram gerados grafos de ocorrências para diversos cenários do modelo. Analisando os relatórios gerados, verificamos que todos os comportamentos obtidos eram os esperados, ou seja, o comportamento dos componentes foi exatamente o definido pela especificação do serviço *Bouncer*. Na Figura 4.8 apresentamos o relatório gerado para um cenário constituído de 02 *Hosts*.

As seguintes conclusões foram obtidas após a análise do relatório:

1. O modelo tem uma marcação morta, isto é, tem *deadlock*. Na transição *1665* o serviço

```
 Statistics
------------------------
 Occurrence Graph

    Nodes:  1665
    Arcs:   3746
    Secs:   10
    Status: Full

 Liveness Properties
------------------------

  Dead Markings:  [1665]
  Dead Transitions Instances: None
```

Figura 4.8: Relatório Gerado do Grafo de Ocorrência

pára, pois todos os processos clientes instanciados nos *Hosts* terminaram de executar, liberando a sua licença ou que tiveram os pedidos de licença negados.

2. O servidor *Bouncer* negou as requisições de licença, quando não havia licença disponível na sua base de dados—Figura 4.9.

```
Bouncer'Servidor 1 1'(("turing","active",
"atim",[("atim",0),("turing",0)]),
("","",("","",[])),("bou","cli",
("lic denied","",[])))
```

Figura 4.9: Servidor Bouncer – *Lic denied*

3. Todas as requisições de serviços foram enviadas e recebidas em uma mesma ordem.

4. Quando não tinha nenhum processo cliente sendo gerenciado pelo servidor *Bouncer* ele tornava-se inativo—Figura 4.10 .

```
Bouncer'Servidor 4  1'(("dunga","inactive","",
[]),("","",("","",[])),("","",("","",[])))
```

Figura 4.10: Servidor Bouncer – *Inactive*

5. Somente um *Bouncer* foi iniciado para cada *Host*.

```
Host'1 1'("atim","cli","bouncer ok")
Host'2 1'("turing","cli","bouncer ok")
Host'3 1'("zabumba","cli","bouncer ok")
Host'4 1'("dunga","","inactive")
```

Figura 4.11: Host – um *Bouncer* Iniciado

6. Todos os servidores Bouncers ativos na rede pertenciam ao grupo bouncer e todas as bases de dados locais eram consistentes (Figura 4.9).

7. É sempre possível retornar a marcação inicial do sistema.

4.2 Estudo Comparativo de Modelagem

Um estudo comparativo de modelagem pode ser efetuado usando diversas métricas. Não existe, portanto, uma maneira específica para realizar esta tarefa. Para efetuarmos a comparação de dois programas, por exemplo, podemos verificar o número de linhas de código, tempo de processamento, etc. Inicialmente, pretendíamos efetuar uma análise dos produtos de modelagem levando em consideração o tamanho dos modelos, legibilidade e identificação com o sistema de software real. No decorrer do desenvolvimento do trabalho resolvemos não efetuar o estudo analisando o tamanho dos modelos, pois o projetista é livre para detalhar as ações de um componente como desejar, o que pode levar a um modelo com tamanhos variados. Optamos por fazer a comparação entre os modelos obtidos, levando em consideração a legibilidade e identificação com o sistema de software real. Os resultados das comparações efetuadas foram:

1. No modelo HCPN, a página de *hierarquia* (Figura 4.1) não evidencia os componentes do sistema e seus relacionamentos. Ainda que os componentes do sistema estejam presentes, seus relacionamentos não são expressos em termos significativos em relação ao sistema real. A informação é mais estrutural do que ligada às possibilidades de comportamento do sistema. As relações entre os componentes são feitas para serem estáticas, qualquer dinamicidade é escondida nas redes que detalham as páginas do modelo. Essa descrição de mais alto nível é especialmente voltada para as relações estruturais (visuais) da organização funcional do modelo. O modelo em HCPN enfatiza

a forma como as redes são compostas para formar um modelo coeso em redes de Petri ao invés de registrar as relações semânticas entre os componentes. No modelo RPOO, as relações ficam mais evidentes.

2. No modelo HCPN é difícil manter o encapsulamento das informações, pois alguns lugares são compartilhados em várias páginas. As fichas depositadas nestes lugares podem ser consultadas e alteradas em qualquer uma destas páginas. Com o acesso de informações em diversas páginas, pode-se perder o controle das mesmas.

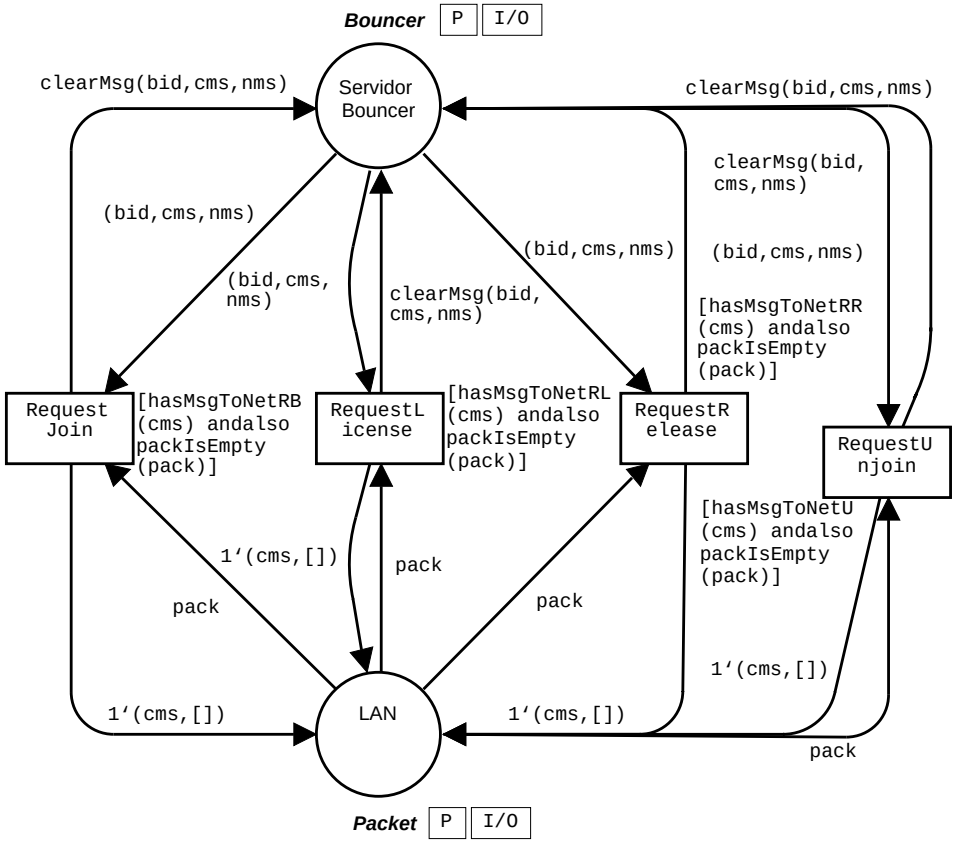


Figura 4.12: Página SendServiceGC

3. A linguagem HCPN não oferece mecanismos para efetuar a troca direta de mensagem entre os componentes. Como não existe uma regra para efetuar a comunicação entre os componentes, para prover este mecanismo de forma mais direta, optamos pela utilização de funções (o que na maioria das vezes, não permite a compreensão das informações trocadas entre os componentes, além de sobrecarregar as inscrições de arco do modelo). Durante o processo de desenvolvimento do modelo, tivemos dificuldade em modelar a comunicação entre os componentes, principalmente na página

do serviço de comunicação em grupo. Para garantir a troca correta das mensagens de requisições de serviços, foram necessárias várias funções, distribuídas nas guardas das transições e inscrições de arco do modelo—Figura 4.12, o que resultou em alguns erros de codificação. No modelo RPOO, a classe *SERVICEGC*—Figura 4.13, que representa o serviço de comunicação em grupo, foi modelada mais facilmente. As trocas de mensagens foram efetuadas através das inscrições de interação, facilitando o envio de mensagens aos objetos do sistema. A comunicação entre os componentes do sistema em RPOO é evidente.

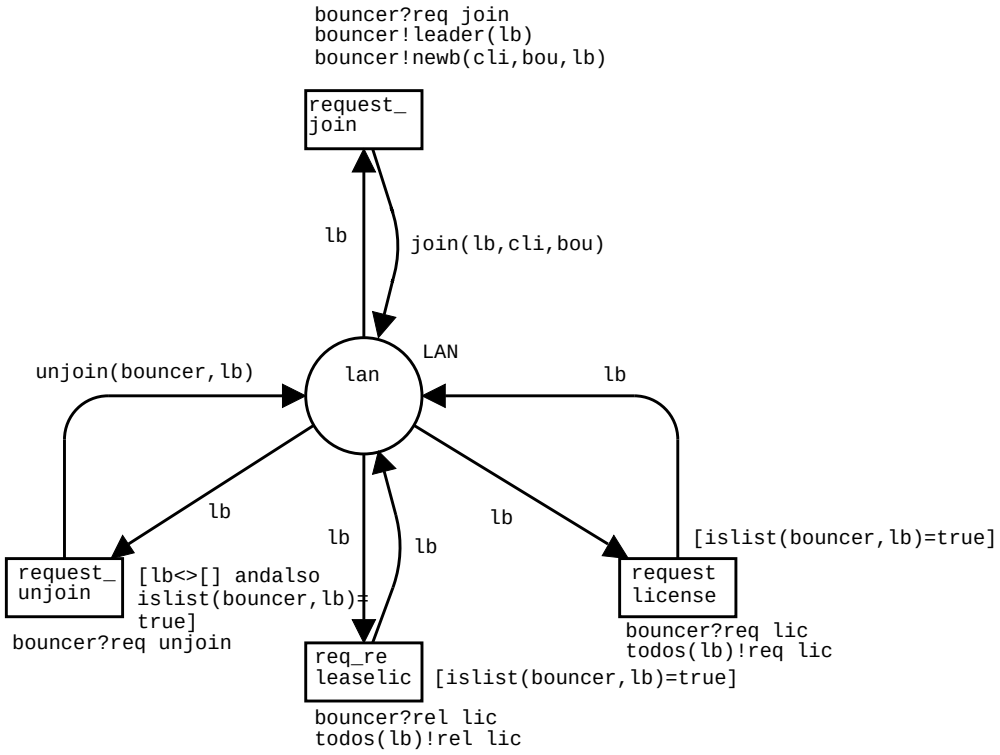


Figura 4.13: Classe SERVICEGC

4. Devido HCPN usar um mecanismo hierárquico para decompor e estruturar os modelos, nem sempre é possível submeter as páginas a análise local. Os níveis mais abstratos não podem ser considerados redes de Petri no sentido formal. As transições de substituição não possuem a mesma semântica das transições normais, elas não consomem e nem devolvem fichas, o que impossibilita a aplicação dos métodos de análise (Figura 4.14). Do ponto de vista formal, a página mais abstrata e todas as redes que detalham as transições de substituição podem ser submetidas à análise se substituirmos

as referidas transições *From Client*, *To Client* e *GCService* por suas redes equivalentes. O que o modelador ganha com a decomposição é perdido na hora da análise, porque o modelo é transformado como se fosse uma única rede. Como as transições de substituição não são uma abstração comportamental, é somente uma abstração visual (não é uma abstração comportamental da outra rede), é inibido as possibilidades de analisar os níveis mais abstrato como se fossem uma rede de Petri.

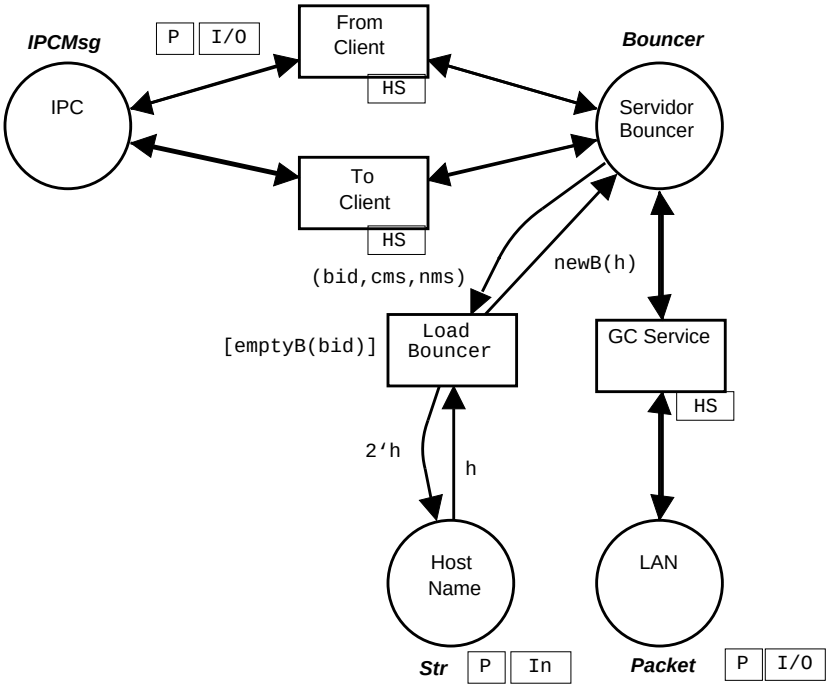


Figura 4.14: Página *Bouncer*

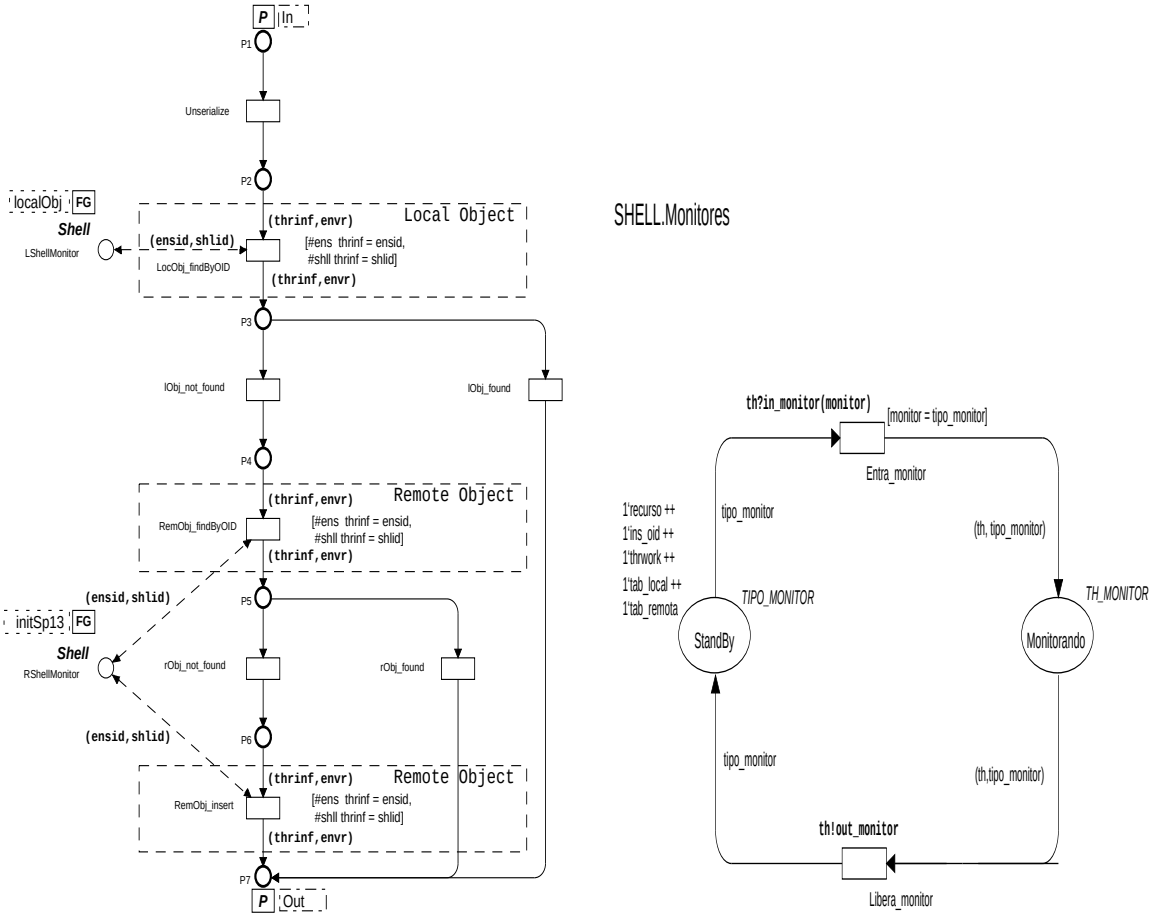
Não existe uma ferramenta de suporte computacional para efetuar análise nos modelos RPOO. Entretanto, existem resultados que provêem uma forte base para o desenvolvimento de análises locais [Gue02]. Esses resultados tem permitido desenvolver algumas análises locais semi-automáticas, usando a ferramenta Design/CPN.

4.2.1 Outras Considerações

O estudo comparativo realizado para o sistema *DistBeta* e protocolo *OSPF* consolidou os itens descritos na seção anterior. Nesta seção apresentamos algumas considerações adicionais para ambos os experimentos.

No modelo HCPN do sistema *DistBeta* os modeladores utilizaram outro mecanismo de

estruturação, que são os lugares de fusão. É difícil manter o encapsulamento das informações tratadas pelo sistema, pois os lugares de fusão *LShellMonitor* e *RShellMonitor* são compartilhados em várias páginas—Figura 4.15(a). As fichas depositadas nestes lugares podem ser consultadas e alteradas em qualquer uma destas páginas. Assim, o comportamento de cada rede pode ser alterado pela rede em que os lugares são utilizados. Com o acesso de informações em diversas páginas, pode-se perder o controle de acesso aos monitores. Estes lugares de fusão armazenam os monitores do modelo. No modelo RPOO, os monitores estão representados em uma única página (Figura 4.15(b)), tornando-o mais modularizado.

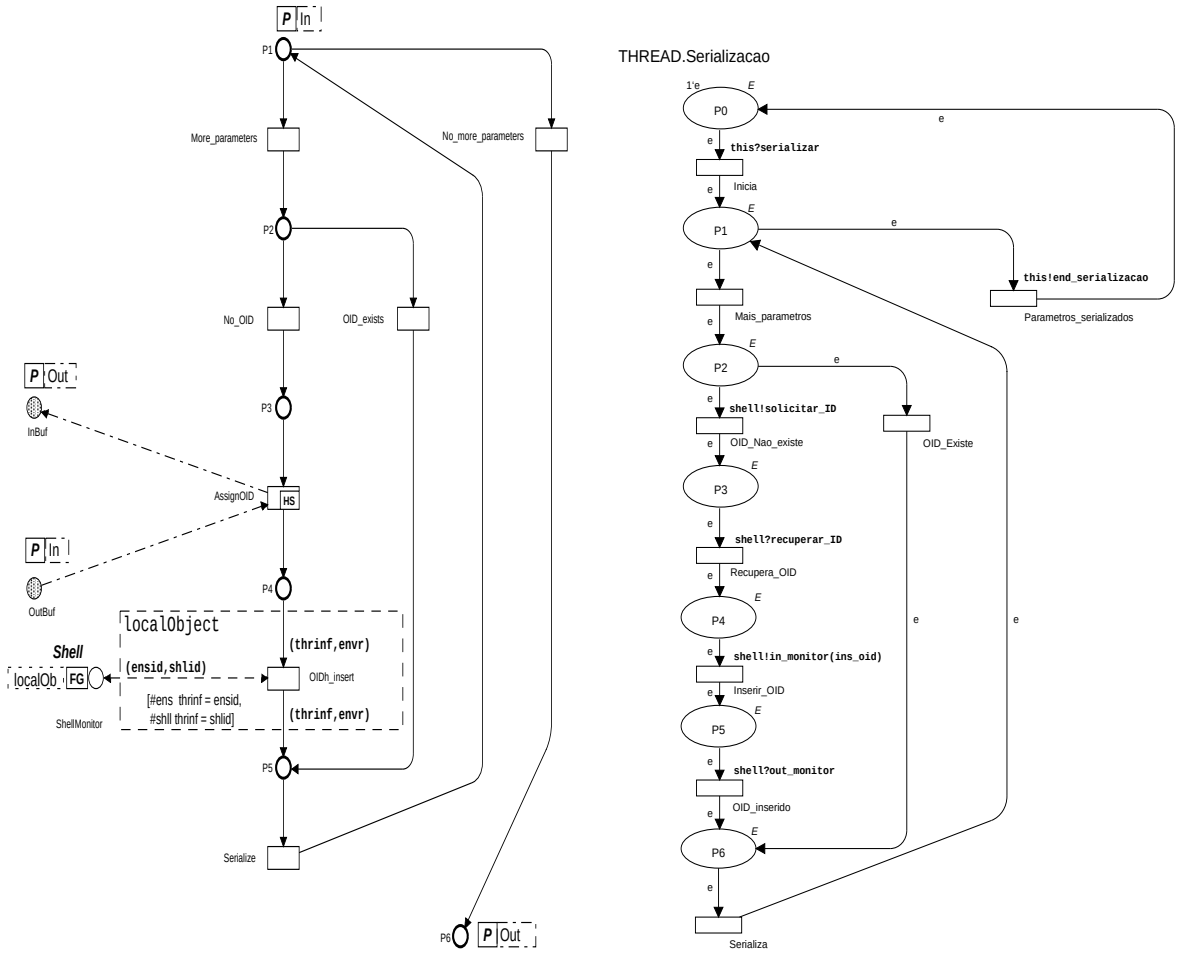


(a) Página Get (b) Classe SHELL_Monitores

Figura 4.15: Página Get x Classe SHELL_Monitores

Nos produtos de modelagem do sistema *DistBeta* a serialização de parâmetros foi modelada utilizando transições de substituição. O comportamento está em duas páginas, o que dificulta o entendimento das ações do *user thread* para tal atividade (Figura 4.16(a)). No

experimento usando RPOO, a serialização de parâmetros foi modelada em uma única página (Figura 4.16(b)), o que permite identificar todas as ações realizadas na serialização de parâmetros.



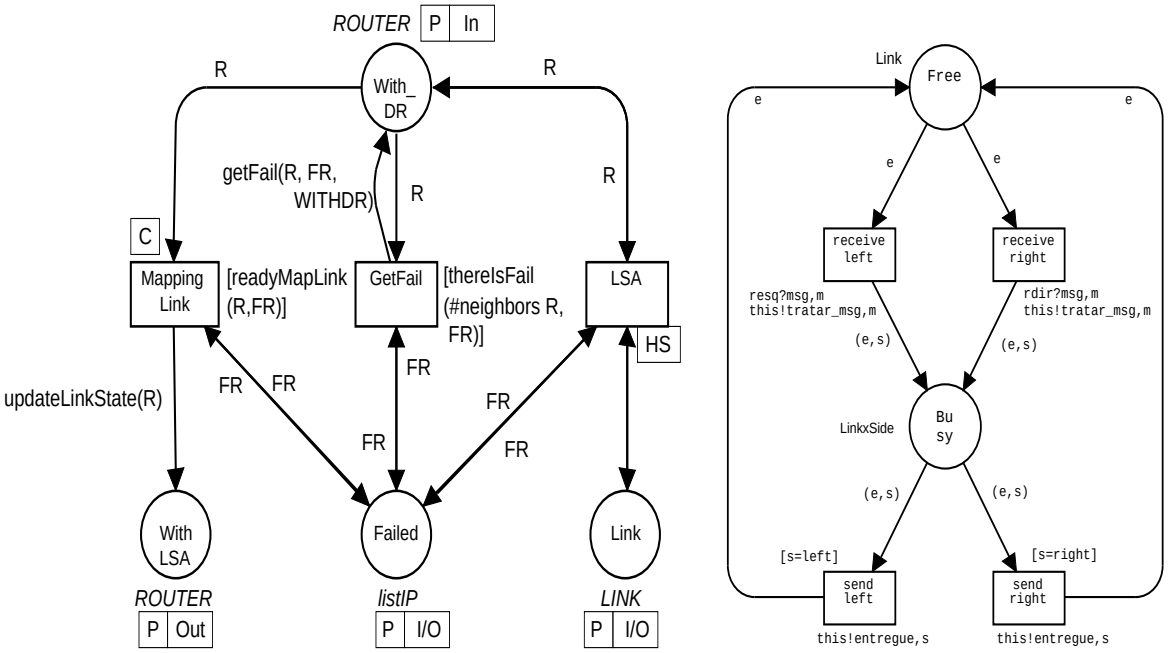
(a) Página Put (b) Classe THREAD_Serialização

Figura 4.16: Página Put x Classe THREAD_Serialização

No modelo HCPN do protocolo *OSPF*, o componente *Link* está presente em diversas páginas, tornando-o menos modularizado—Figura 4.17(a). No modelo RPOO, este componente foi modelado separadamente no corpo da classe *LINK* – Figura 4.17(b).

Os modelos completos do Sistema *DistBeta* e do protocolo *OSPF* utilizando a linguagem HCPN encontram-se nos Apêndices B e C, respectivamente.

Embora o protocolo *OSPF* não atendesse as características desejáveis da notação RPOO, os produtos de modelagem obtidos representam adequadamente as funcionalidades do pro-



(a) Página Mapping_Link (b) Classe LINK

Figura 4.17: Página Mapping_Link x Classe LINK

tocolo.

A comparação entre diferentes soluções não é uma tarefa fácil. Sabemos que é necessário elaborar mais experimentos de comparação, mas estes experimentos iniciais nos deram uma boa indicação que RPOO pode ser aplicada na modelagem de vários tipos de sistemas.

Capítulo 5

Conclusão

Ao longo deste trabalho, apresentamos o uso da linguagem RPOO aplicado a um contexto prático para decompor e estruturar sistemas distribuídos e concorrentes. Inicialmente, as abordagens redes de Petri coloridas, redes de Petri coloridas hierárquicas e RPOO foram discutidas. Em seguida, apresentamos os experimentos de modelagem utilizando a linguagem RPOO, bem como uma análise dos modelos. Finalmente, apresentamos a modelagem dos experimentos utilizando HCPN e um estudo comparativo dos produtos de modelagem obtidos usando as abordagens HCPN e RPOO. Além dos experimentos desenvolvidos, foram conduzidas algumas simulações e análises dos modelos utilizando as ferramentas Design/CPN e SSO.

Os resultados das simulações foram visualizados através dos diagramas de sequência de mensagens (MSC's). As simulações efetuadas possibilitaram a verificação do comportamento dos objetos do sistema, aumentando a nossa confiança na corretude dos modelos RPOO. Durante as simulações o comportamento dos objetos instanciados foram exatamente os definidos pelas especificações dos experimentos.

Durante este trabalho efetuamos um estudo da linguagem RPOO, onde verificamos que a mesma é de fácil compreensão para desenvolvedores fluentes em redes de Petri e orientação a objetos, pois o tempo dedicado ao aprendizado da notação foi pequeno. A maior parte do tempo dedicado ao desenvolvimento dos produtos de modelagem foi gasto na compreensão da especificação dos experimentos e na modelagem dos mesmos. Dentre as facilidades oferecidas pela notação, está a modelagem da comunicação entre os objetos do sistema. Todas as trocas de mensagens entre os objetos foram efetuadas através das inscrições de interação

associadas às transições dos corpos das classes do modelo.

Os experimentos de modelagem apresentados nos permitiu experimentar e validar a sintaxe definida pela notação. As alterações de sintaxe sugeridas foram incorporadas na notação. Além disso, a sintaxe foi suficiente para modelar os experimentos escolhidos, inclusive o terceiro experimento, que não pertence à classe de sistemas para a qual a notação foi desenvolvida.

Com este trabalho temos exemplos do uso da notação RPOO aplicado a diversos cenários, o que contribui com a documentação de RPOO, proporcionando exemplos com uma descrição detalhada que poderão servir de guia/suporte a outros modeladores durante a construção de modelos RPOO. Além disso, esse trabalho propõe uma metodologia inicial para desenvolver a modelagem de sistemas usando a notação RPOO. O detalhamento da metodologia utilizada encontra-se na apresentação do primeiro experimento de modelagem descrito no Capítulo 3. A maneira como os conceitos de redes de Petri foram integrados com orientação a objetos permite ao modelador que possui conhecimento prévio de orientação a objetos adotar uma certa disciplina para a elaboração de modelos RPOO.

Como parte do processo de validação da linguagem RPOO efetuamos um estudo comparativo de modelagem entre os produtos obtidos em RPOO com os produtos desenvolvidos usando o formalismo HCPN. Esta comparação foi efetuada porque a linguagem HCPN é o padrão estabelecido para decompor e estruturar sistemas em redes e Petri.

A aplicação da notação na modelagem de sistemas reais nos deu indícios de que ela pode ser aplicada a diversos tipos de sistemas, pois a mesma demonstrou ser uma boa alternativa para o problema da estruturação e decomposição de sistemas em redes de Petri.

Um processo natural no desenvolvimento de uma linguagem de modelagem é a construção de ferramentas de suporte computacional. Os experimentos contribuíram na definição do conjunto de ferramentas necessárias para RPOO. A construção de ferramentas para suportar a modelagem, análise e validação de modelos RPOO são de extrema importância para a aplicação da notação. Uma versão inicial de uma ferramenta de simulação para RPOO foi desenvolvida. Os experimentos desenvolvidos contribuíram na fase de teste da versão inicial desta ferramenta. A utilização da ferramenta possibilitou a validação dos modelos RPOO desenvolvidos para os experimentos.

Bibliografia

[BB88] M. Baldassari and G. Bruno. An environment for object-oriented conceptual programming based on prot nets. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 340 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 1988.

[BBF97] Tárcio Rodrigues Bezerra, Francisco Vilar Brasileiro, and Walfredo Costa Cirne Filho. Bouncer: Um serviço distribuído e tolerante a faltas para controle de licenças de software. In Marcos Borges, editor, *VII Simpósio de Computadores Tolerantes a Falhas*, pages 221–235, Campina Grande, PB – Brasil, July 1997. Sociedade Brasileira de Computação.

[Bed95] Marek A. Bednarczyk. A comparison of object-oriented development methodologies. Url:http://www.ipipan.gda.pl/ marek/objects/toa/oomethod/mcr.html, 1995.

[Bez96] Tárcio Rodrigues Bezerra. Bouncer-uma solução distribuída para controle de licenças de software. Dissertação de mestrado, Universidade Federal da Paraíba, 1996.

[Car02] Carina Machado de Farias and Jorge César Abrantes de Figueiredo. Uma modelagem do protocolo ospf usando redes de petri hierárquicas. Technical Report DSC/004/2002, September 2002.

[CJK97] S. Christensen, J. B. Joergensen, and L. M. Kristensen. Design/CPN — A computer tool for coloured Petri nets. *Lecture Notes in Computer Science*, 1217:209, 1997.

[CKR94] L. Cherkasova, V. Kotov, and T. Rokicki. On net modelling of industrial size concurrent systems. In *Proceedings of 15th International Conference on the Application and Theory of Petri Nets*, Zaragoza, 1994.

[DCdFP93] Y. Deng, S.K. Chang, J.C.A. de Figueiredo, and A. Perkusich. Integrating software engineering methods and petri nets for the specification and prototyping of complex software systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 206 – 223. Springer-Verlag, Chicago, USA, jun 1993.

[Dij] Algoritmo de dijkstra para cálculo do caminho de custo mínimo. Url:http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html.

[D.U94] Jeffrey D.Ullman. *Elements of ML Programming*. Prentice Hall, 1994.

[EJ96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, September 1996. ftp://reports.adm.cs.cmu.edu/usr/anon/1996/CMU-CS-96-178.ps.

[ELR90] J. Engelfriet, G. Leih, and G Rozenberg. Net-based description of parallel object-based systems, or POTs and POPs. pages 229–273, 1990.

[Gen87] H.J. Genrich. Predicate/Transition nets. In W. Brauer, W. Reisig, and G. Rozemberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, 1987.

[Gue97] Dalton Dario Serey Guerrero. Sistemas de redes de petri modulares baseadas em objetos. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, 1997.

[Gue98] Dalton D. S. Guerrero. Orientação a objetos e modelos de redes de petri. Technical report, Coordenação de Pós-graduação em Engenharia Elétrica - COPELE/UFPB, Campina Grande, PB, April 1998.

[Gue02] Dalton Dario Serey Guerrero. Redes de petri orientadas a objetos. Tese de doutorado, COPELE - Universidade Federal da Paraíba, 2002.

[Hay99] David C. Hay. A comparison of data modeling techniques. Url:http://www.essencialstrategies.com/publications/modeling/compare.htm, 1999.

[HBG93] Shuguang Hong, Sjaak Brinkkemper, and Geert Van Den Goor. A formal approach to the comparison of object-oriented analysis and design methodologies. pages 689–698, Hawaii, January 1993. 26th Hawaii International Conference of System Sciences.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Jen] Kurt Jensen. Simple protocol. Url:http://www.daimi.aau.dk/designcpn/exam/examples/simpleprotocol/index.html.

[Jen92a] K. Jensen. *Coloured Petri Nets 3: Basic Concepts, Analysis Methods and Practical Use*, volume 3. Springer-Verlag, Berlin, Alemanha, 1992.

[Jen92b] Kurt Jensen. *Coloured Petri Nets 1: Basic Concepts, Analysis Methods and Pratical Use, volume 2*. Springer-Verlag,Berlin,Germany, 1992.

[Jen92c] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis, Methods and Practical Use, Volume 1*. EACTS – Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[JM95] J. B. Jorgensen and K. H. Mortensen. Modelling and Analysis of Distribuited Program Execution in BETA Using Coulored Petri Nets. Technical report, 1995.

[Lak95] Charles Lakos. From Coloured Petri nets to object Petri nets. In *Proceedings of the 15th International Conference on the Application and Theory of Petri Nets*, Lecture Notes in Computer Science, pages 278–297. Springer Verlag, Turin, Italy, 1995.

[Lar98] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, 1998.

[MAM95] G. Conte S. Donatelli e G. Franceschinis M. Ajmone Marsan, G. Balbo. *Modelling with Generalized Stochastic Petri Nets*. John Wiley & Sons, 1995.

[MSC] Design/CPN message sequence charts library. Url: http://www.daimi.au.dk/designcpn/libs/mscharts.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, April 1989.

[Pai99] Richard F. Paige. A comparison of the business object notation and the unified modeling language. volume 1723 of *Lecture Notes in Computer Science*, pages 67–82. Second International Conference de UML, Springer-Verlag, October 1999.

[Rfc] Request for comments 1583. Url:http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1583.html.

[RJB98] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series. ACM Press and Addison-Wesley, 1998.

[San01] José Amancio Macedo Santos. Ferramentas de suporte à simulação de rpoo. Proposta de dissertação de mestrado, COPIN – Universidade Federal da Paraíba, 2001.

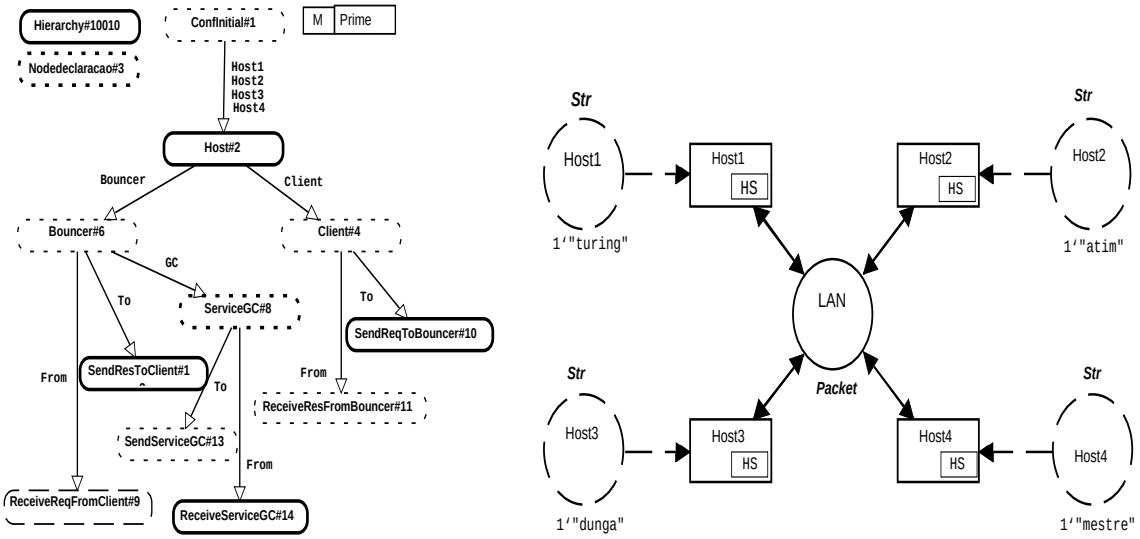
[Tan97] Andrew S. Tanenbaum. *Redes de Computadores*. Campus, 1997.

[THC90] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 1990.

Apêndice A

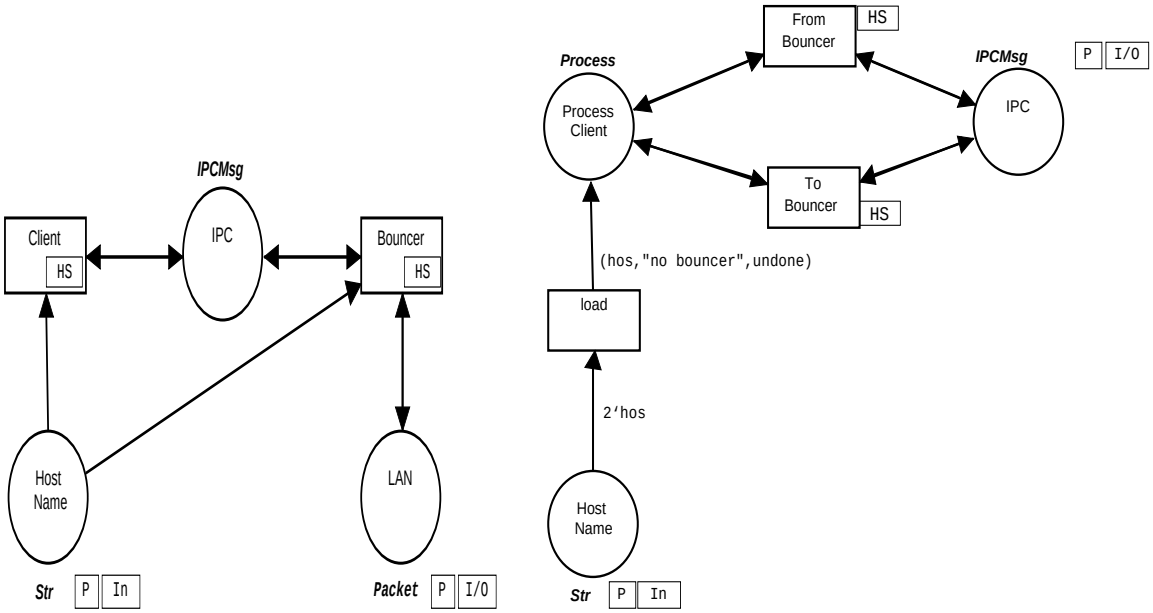
Primeiro Experimento de Modelagem

A Serviço *Bouncer* – Modelo Completo Utilizando HCPN e RPOO



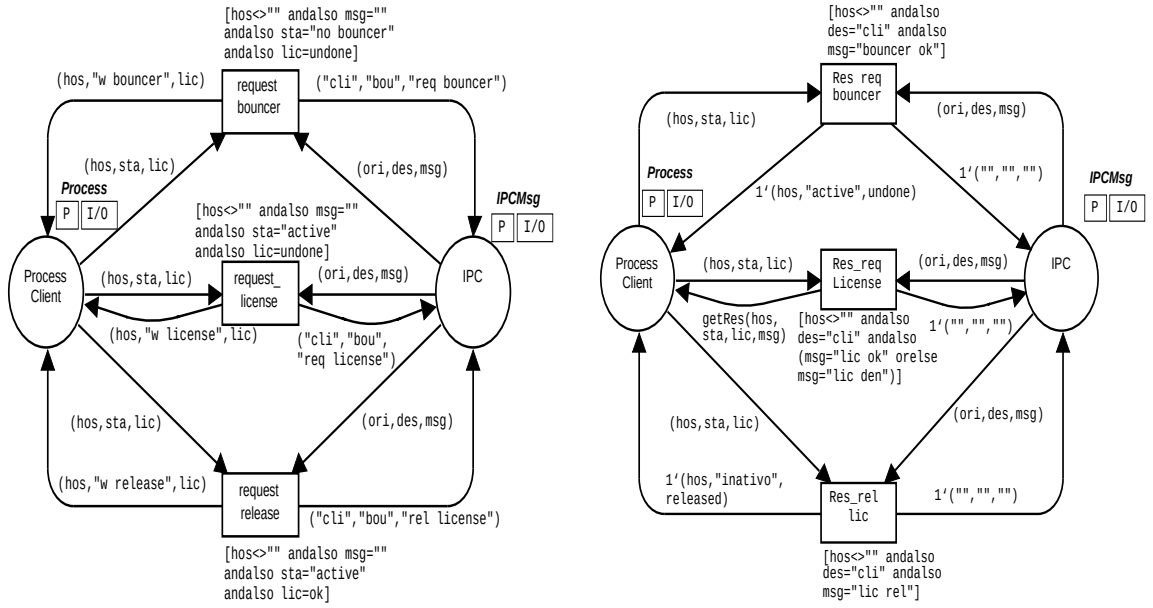
(a) Página de *Hierarquia* (b) Página *ConfInitial* - página principal do Bouncer

Figura A.1: Página de *Hierarquia* e Página *ConfInitial*



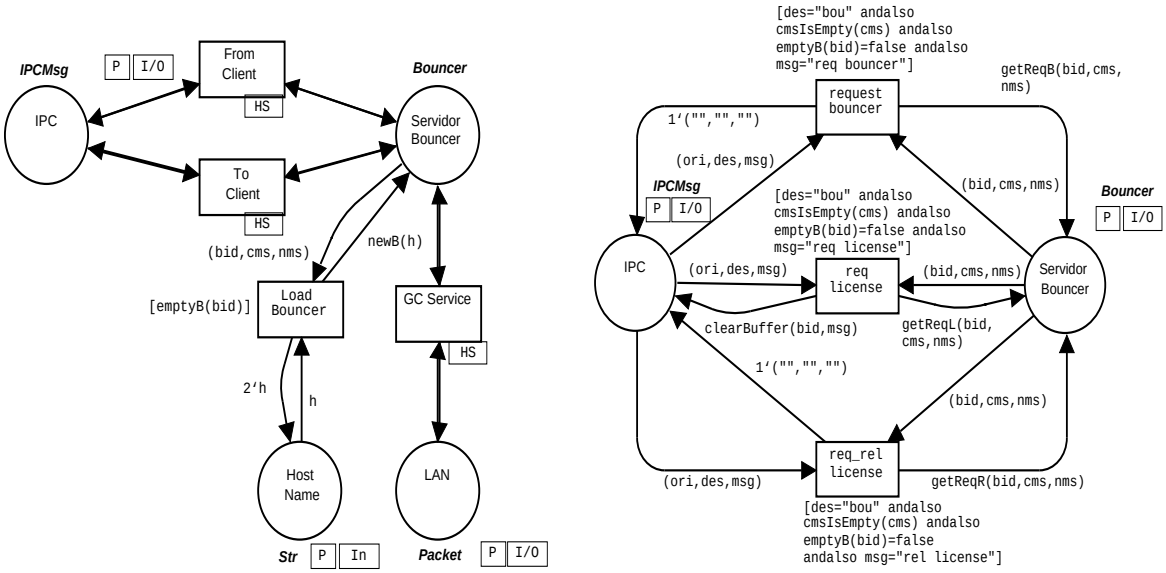
(a) Página *Host* (b) Página *Client*

Figura A.2: Página *Host* e página *Client*



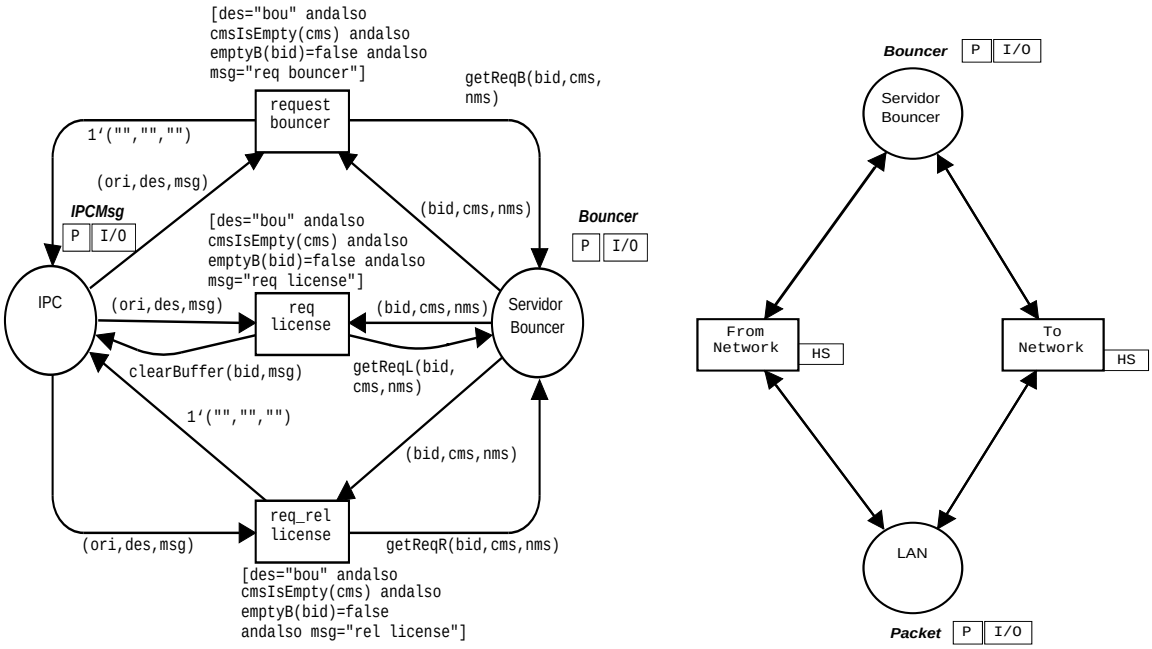
(a) Página *SendReqToBouncer* (b) Página *ReceiveResFromBouncer*

Figura A.3: Página *SendReqToBouncer* e Página *ReceiveResFromBouncer*



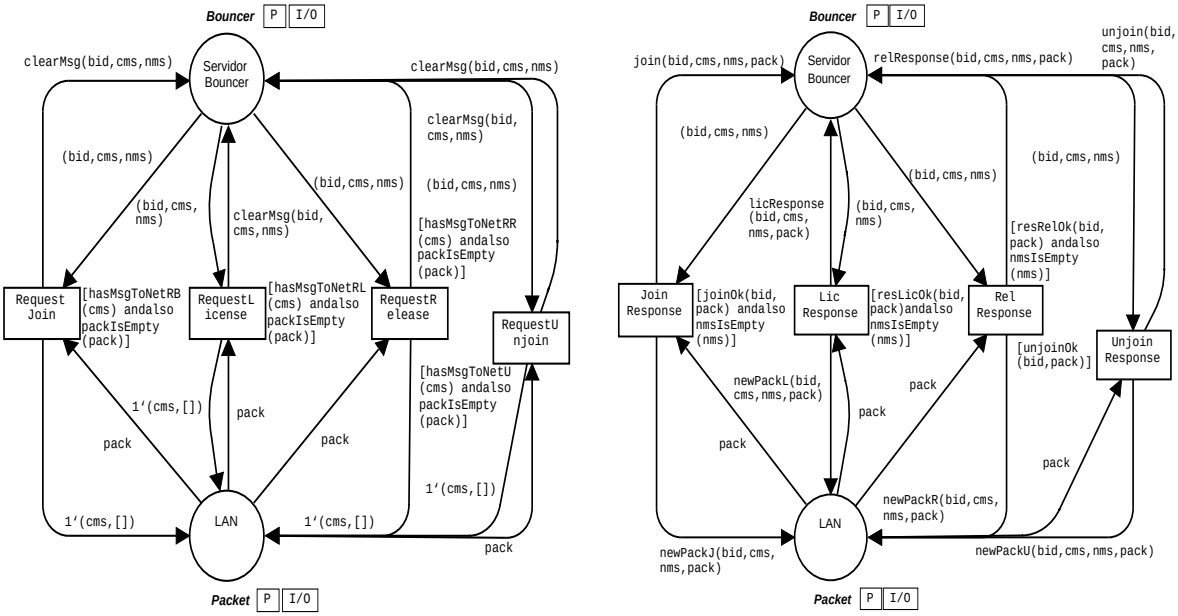
(a) Página *Bouncer*–Servidor Bouncer (b) Página *ReceiveReqFromClient*

Figura A.4: Página *Bouncer* e Página *ReceiveReqFromClient*



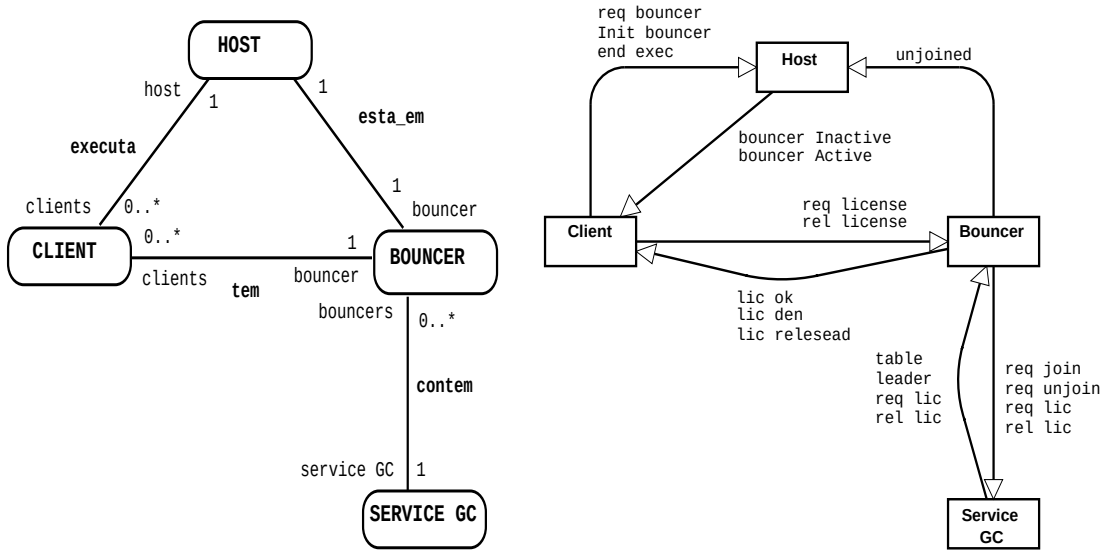
(a) Página *SendResToClient* (b) Página *ServiceGC*

Figura A.5: Página *SendResToClient* e Página *ServiceGC*



(a) Página *SendServiceGC* (b) Página *ReceiveServiceGC*

Figura A.6: Página *SendServiceGC* e Página *ReceiveServiceGC*



(a) Diagrama de Classes (b) Diagrama de Fluxo de Mensagens

Figura A.7: Diagrama de Classes e Diagrama de Fluxo de Mensagens

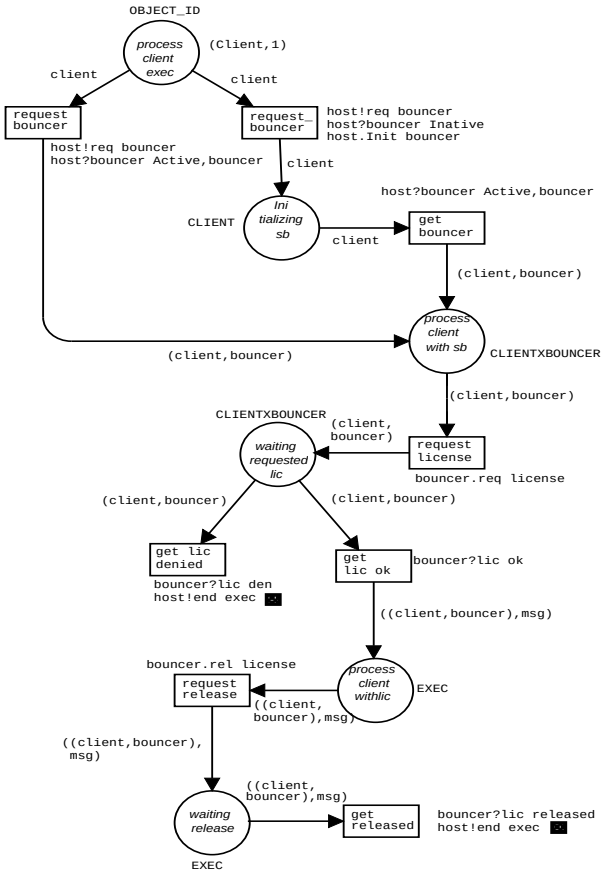
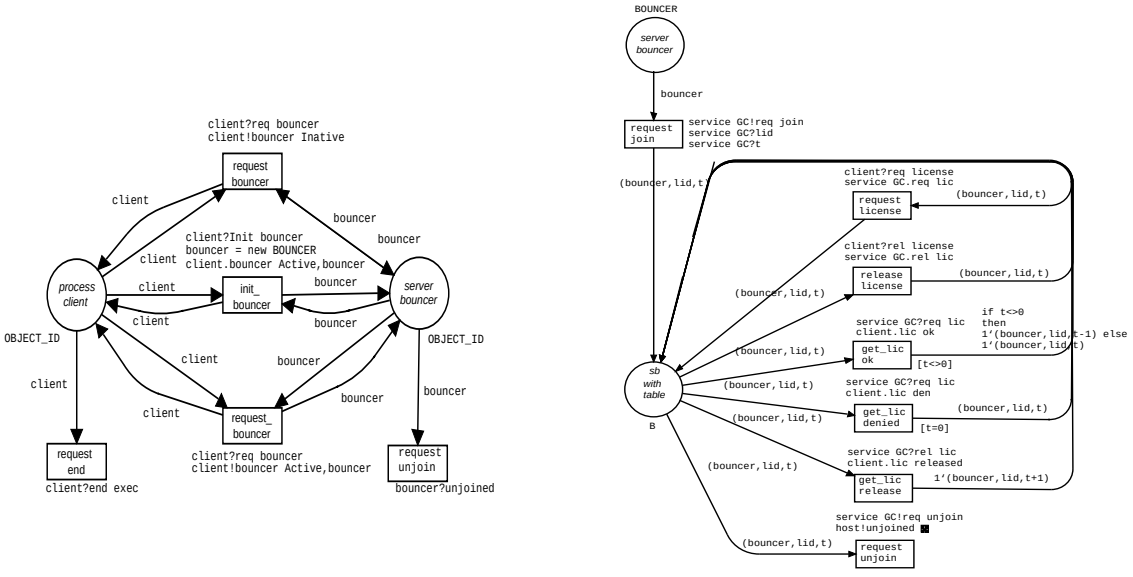


Figura A.8: Classe CLIENT



(a) Classe HOST

(b) Classe BOUNCER

Figura A.9: Classe HOST e Classe BOUNCER

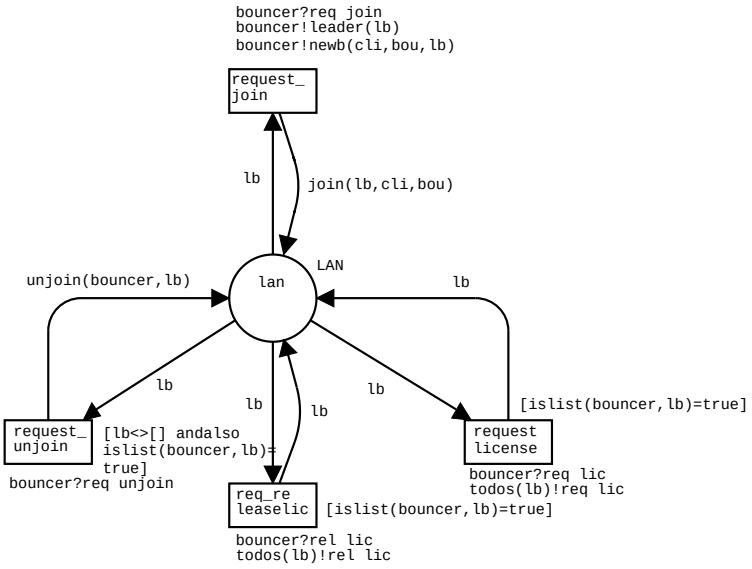


Figura A.10: Classe SERVICEGC

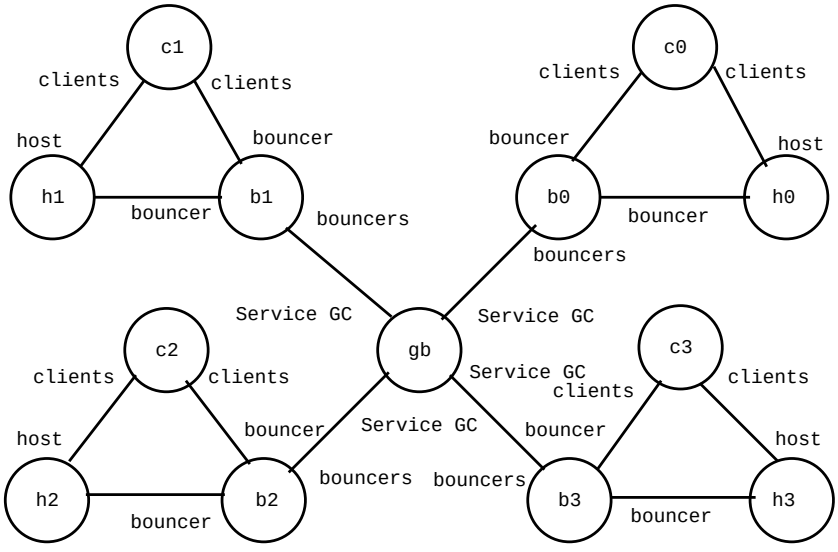


Figura A.11: Diagrama de Configuração Inicial

Apêndice B

Segundo Experimento de Modelagem

B Sistema *DistBeta* – Modelo HCPN e RPOO

Os produtos de modelagem do Sistema *DistBeta* utilizando HCPN foram desenvolvidos por um grupo de modeladores [JM95; Jen92a], constituído de 02 membros, do Departamento de Ciência da Computação da Universidade de Aarhus—Dinamarca, experientes em modelagem de sistemas com HCPN.

B.1 Página de Hierarquia

Na Figura B.1 é apresentada a página de hierarquia do modelo, que possui um total de 15 páginas que detalham o comportamento do sistema. O modelo consiste de cinco partes:

- Descrição Geral—página *DistBeta*;
- Network—página *Network*;
- Sender—página *Send*;
- Receiver—página *Receive*;
- Inicialização—página *Configur*.

As páginas *Put, AssignOI, RPCCall, Get, Listen, Execute,* e *OIDGen* ajudam a descrever as partes *Sender* e *Receiver* do modelo. Essas partes comunicam-se através da rede – página *Network*.

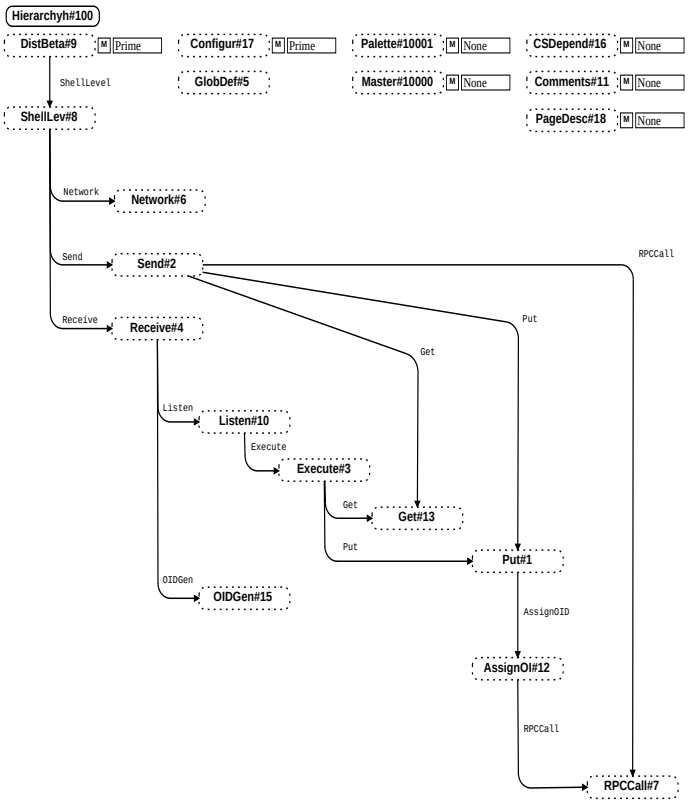


Figura B.1: Página de *Hierarquia*

B.2 Parte de Inicialização

Na Figura B.2 é apresentada a página de configuração inicial do sistema–página *Configur*. Inicialmente, temos apenas a transição *Initialize_Model* habilitada. Esta transição irá ocorrer apenas uma vez. Quando ela disparar, fichas para os *users threads*, *listeners threads*, *shells*, *buffers*, *monitores* e *ensembles* serão geradas em função da variável *model_Config* que define a configuração do modelo e depositadas nos diversos lugares de fusão da rede. Estes lugares estão presentes em várias outras páginas do modelo.

B.3 Parte de Descrição Geral

Na Figura B.3 é apresentada a página de descrição geral—página *DistBeta*. Esta página representa o nível mais abstrato do modelo. Todos os *users threads* iniciados estão no lugar *p1*. O *user thread* deve selecionar o servidor (isto é, um *shell* em um *ensemble*) que contém o objeto com quem ele deseja se comunicar, transição *SelectServer*. Todos os *listener threads* estão inicialmente no lugar de fusão *AllListeners*. A transição *SelectServer* remove

uma ficha do lugar *P1* e uma ficha do lugar *AllListeners* e gera uma ficha no lugar *P2* através da função *assignServer*. A função *assignServer* armazena o *ensemble* em que o objeto chamado está localizado. Tendo o *ensemble* de destino, o próximo passo é fazer a chamada ao objeto remoto. Isto é feito através da transição de substituição *ShellLevel*. Após completar a chamada o *user thread* retorna para o lugar *P1*.

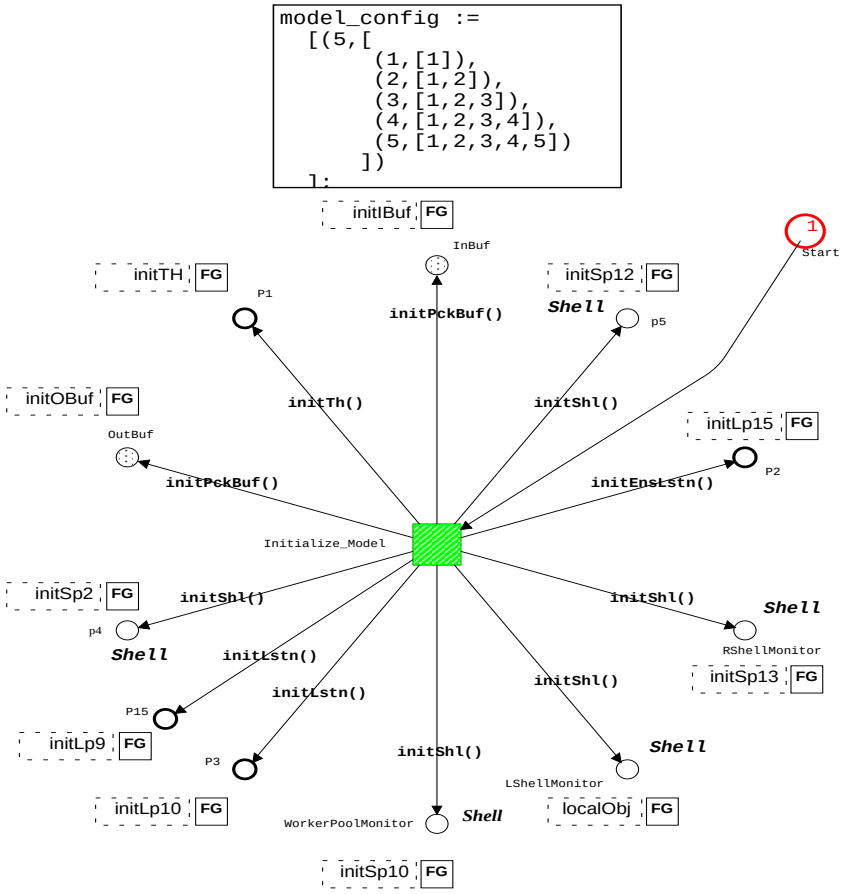


Figura B.2: Página *Configur* – Página de Configuração Inicial

Os Shells tem duas partes: *Send* e *Receive*. As duas partes comunicam-se através da rede—página *Network*.

B.4 Parte Network

Na Figura B.4 é apresentada a página *Network*. Esta página representa o modelo da rede de comunicação entre os componentes modelados. A rede tem três lugares:

- *InBuf*: contém uma ficha para cada *shell*. Cada ficha representa uma lista de pacotes que os *threads* de um *shell* desejam enviar para a rede.

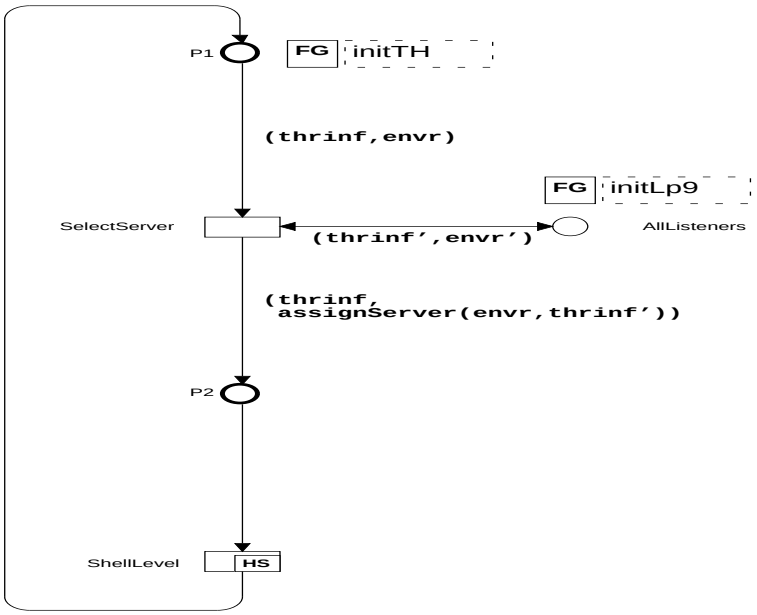


Figura B.3: Página *DistBeta* – Página de mais alto nível

- *Network*: representa uma rede real. Contém sempre uma ficha representando que, ou um pacote está trafegando na rede ou a rede está desocupada.

- *OutBuf*: contém uma ficha para cada *shell*. Cada ficha representa uma lista de pacotes que os *threads* de um *shell* esperam receber.

e três 3 transições:

- *Buf_to_net*: se a rede está desocupada e existe algum pacote a ser transmitido, esta transição ficará habilitada. Ao ser disparada, esta transição irá retirar o primeiro pacote da lista do lugar *InBuf* e colocá-lo na rede *lugar Network*.

- *Net_to_buf*: esta transição vai disparar quando o pacote que vier da rede for destinado ao *shell*. A guarda desta transição garante que o shell de destino seja respeitado. Quando a transição disparar, o pacote será removido da rede e adicionado no lugar *OutBuf*.

- *Network_error*: esta transição modela a possibilidade de ocorrência de erros na transmissão de pacotes na rede. A presença de qualquer ficha (pacote) no lugar *InBuf* habilitará esta transição. Ao ocorrer esta transição, o pacote será removido do lugar *InBuf* e adicionado no lugar *OutBuf*. A mensagem do pacote será modificada para

indicar que ocorreu um erro. A guarda desta transição garante que apenas pacotes de requisições podem ocorrer erros (restrição feita na especificação do sistema), os quais serão tratados na página *receive*.

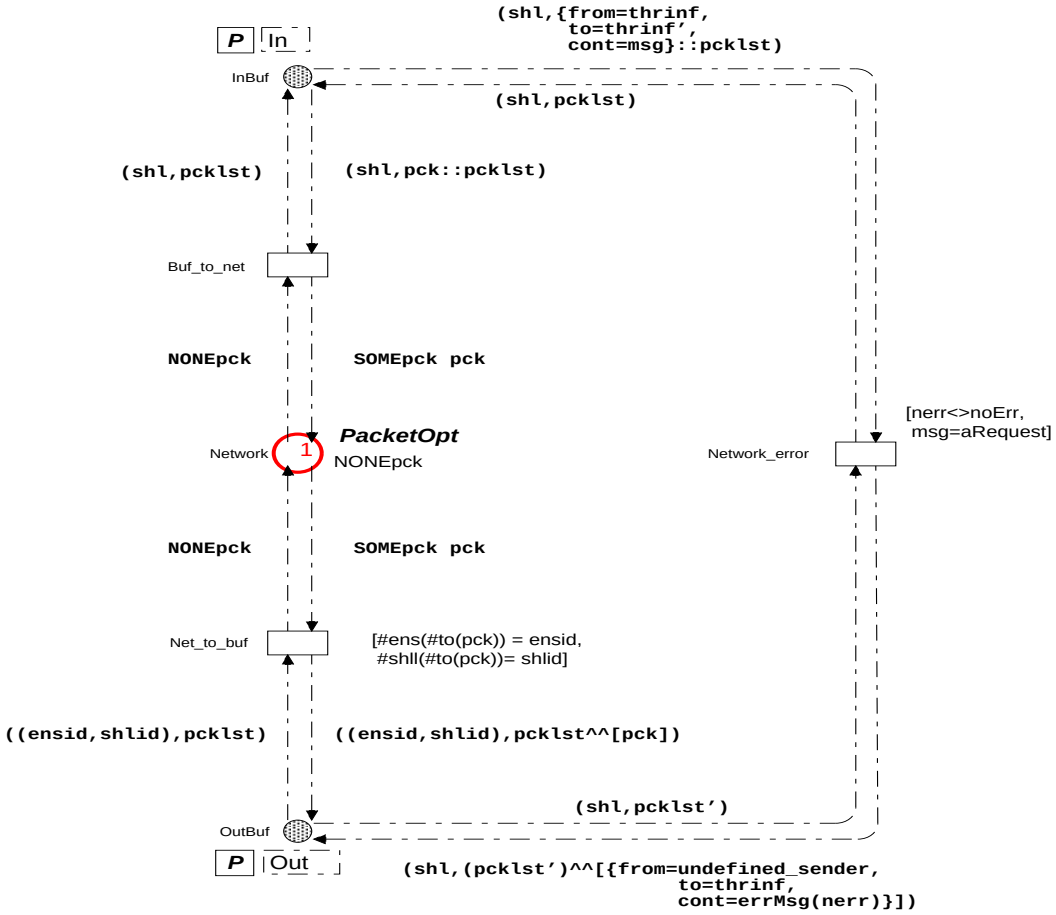


Figura B.4: Página *Network*

B.5 Parte Send

Esta parte é constituída das páginas *Send*,*Put*,*AssignOI*,*RPCCall* e *Get*.

Página Send

Na Figura B.5 é apresentada a página *Send*. Esta página modela as ações básicas do envio de pacotes pelos *users threads*. Inicialmente, o *user thread* obtém os recursos necessários, através da transição *GetResource*. O lugar de fusão *ShellMonitor* contém fichas que controlam

a alocação de recursos. Após obter os recursos necessários, o *user thread* irá fazer a serialização dos parâmetros transição de substituição *Put*. Em seguida, a transição *AssignServer* vai alterar as informações de ambiente do *user thread*, indicando o *user thread* de origem, o *user thread* de destino e que a mensagem é uma requisição. Este procedimento prepara a mensagem de forma apropriada para a chamada RPC. A chamada RPC propriamente dita acontece na transição de substituição *RPCCall*. Se a chamada RPC acontecer sem erros, o *user thread* estará pronto para fazer a desserialização dos parâmetros, que é feita na página que representa a transição de substituição *Get*. A última ação do *user thread* é liberar o recurso obtido no início do processo (transição *GetResource*). A liberação de recursos é protegida por um monitor. Ao liberar os recursos, o *user thread* voltará para o seu estado inicial.

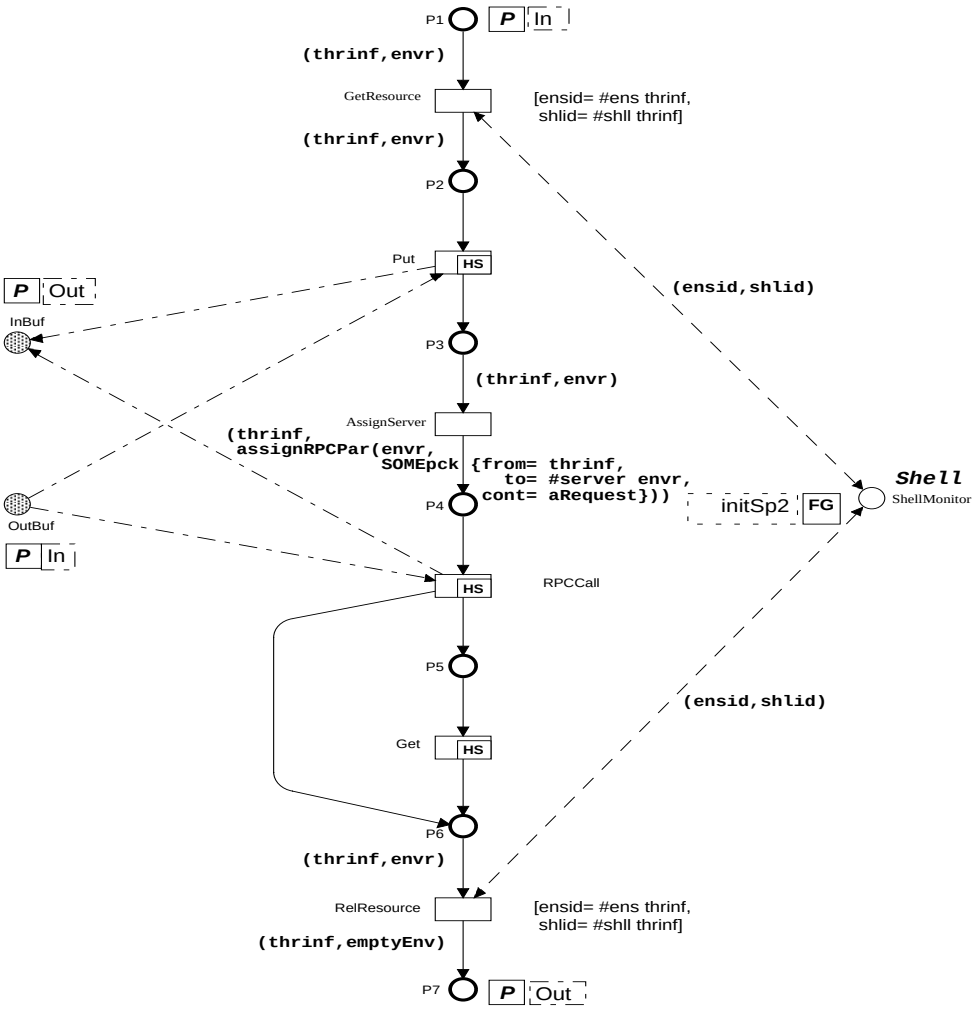


Figura B.5: Página *Send*

Página Put

Na Figura B.6 é apresentada a página *Put*. O objetivo desta página é modelar a serialização dos parâmetros da chamada remota. Inicialmente existem duas situações possíveis:

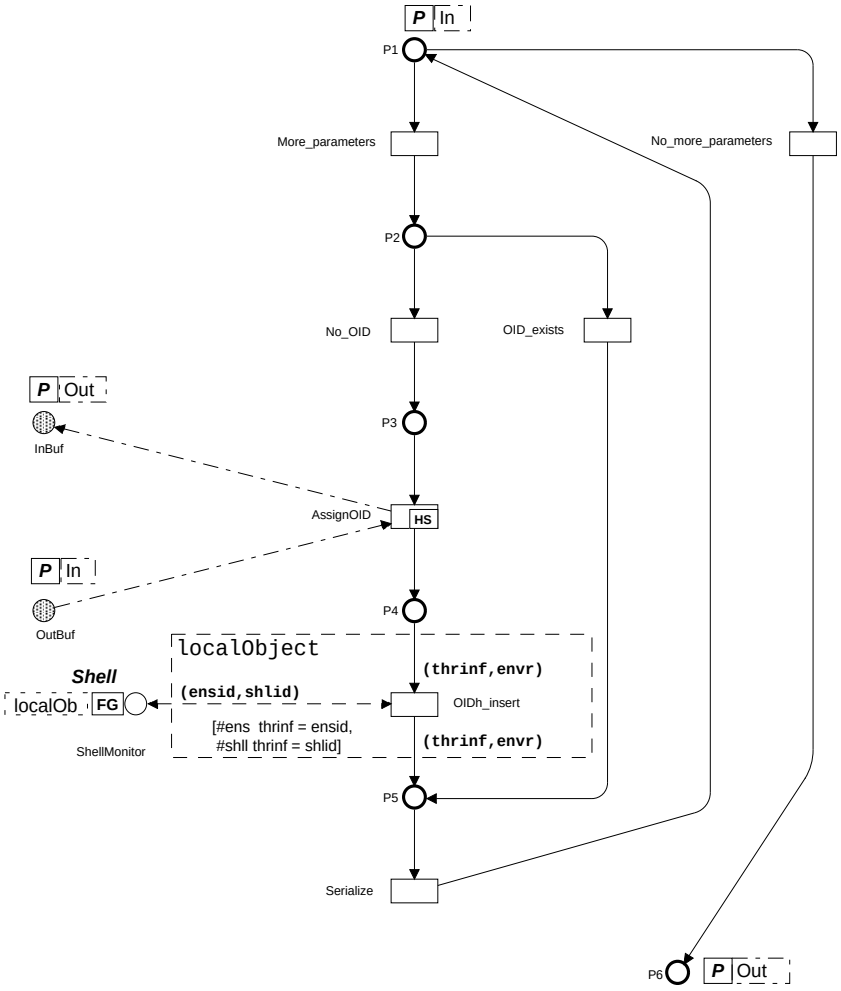


Figura B.6: Página *Put*

1. Existem outros parâmetros que devem ser serializados (transição *More_parameters*).

2. Não existem outros parâmetros (transição *No_more_parameters*).

Se a segunda situação ocorrer, a serialização estará completa e o *user thread* final é colocado no lugar *P6*. Neste estado a chamada remota está pronta para ocorrer. Se a primeira situação ocorrer: caso o objeto parâmetro tratado já possuir um OID, a serialização do parâmetro (transição *Serialize*) poderá ocorrer diretamente. Se o parâmetro não possuir um OID, um OID deve ser obtido (transição de substituição *AssignOID*) e inserido na tabela

que mantém os OID's locais transição *OIDh_insert*. O monitor controla a utilização da tabela de OID (lugar de fusão *ShellMonitor*). Finalmente, o parâmetro considerado pode ser serializado transição *Serialize*.

Página AssignOI

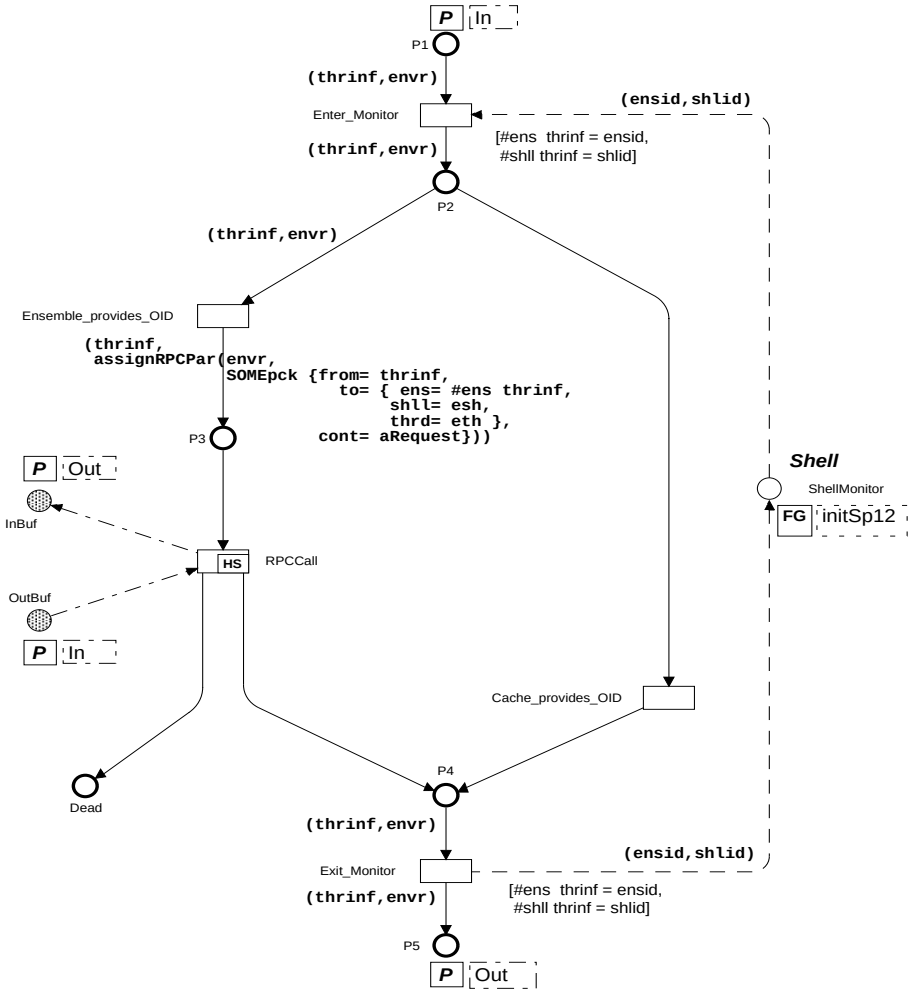


Figura B.7: Página *AssignOI*

Na Figura B.7 é apresentada a página *AssignOI*. Esta página modela a solicitação de um novo OID de objeto por um *user thread*. A obtenção de um novo OID é protegida por um monitor para que dois *users threads* do mesmo *Shell* não possam solicitar ao mesmo tempo um OID e assim evitar inconsistências na tabela de OID. A transição *Enter_Monitor* modela o ponto de entrada neste monitor. Depois que entrar no monitor, o *user thread* poderá solicitar um OID na tabela local (transição *Cache_provides_OID*) ou solicitar ao

Ensemble (transição *Ensemble_provides_OID*). A solicitação de um OID ao *Ensemble* irá gerar uma chamada RPCCall (transição de substituição *RPCCall*), a qual está descrita na página *RPCCall*. A chamada RPCCall neste ponto é semelhante à chamada ocorrida na página *Send*. A única diferença é que aqui não há serialização de parâmetros. Ocorrendo uma chamada RPCCall um erro poderá ser retornado. O tratamento dado aos erros faz com que o *user thread* páre de executar, indo para o lugar *Dead*. Se não ocorrer nenhum erro na chamada RPCCall, o *user thread* poderá sair do monitor, devolvendo a ficha ao lugar *ShellMonitor* e liberando assim o recurso. Se o OID é solicitado ao cache local, a alocação do OID ocorre de forma direta e o *user thread* pode sair normalmente do monitor, como na situação anterior.

Página RPCCall

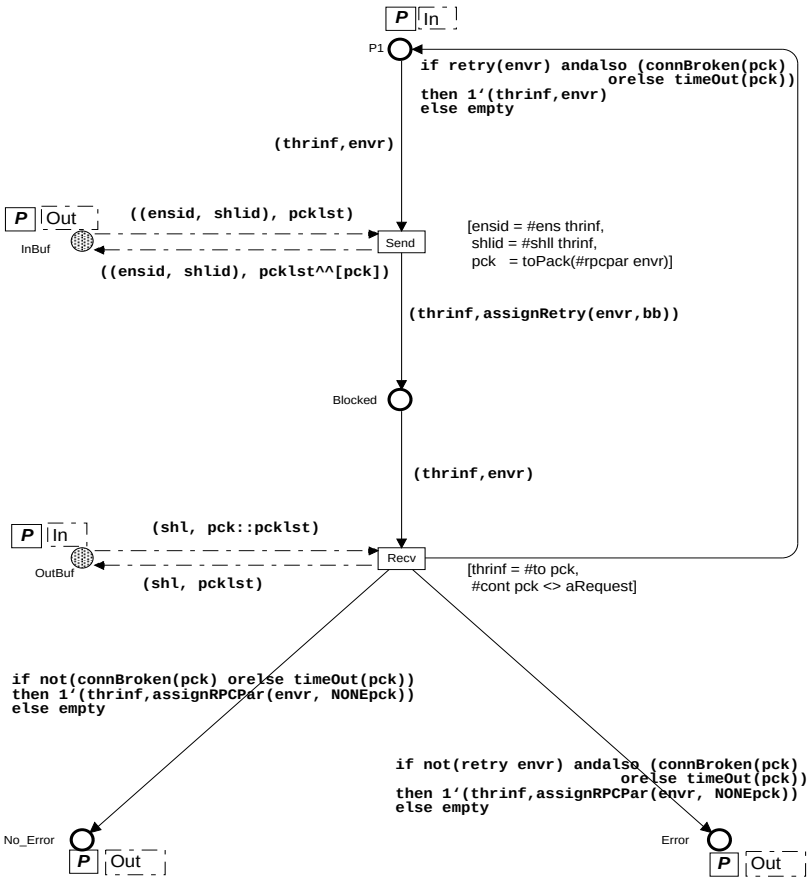


Figura B.8: Página *RPCCall*

Na Figura B.8 é apresentada a página *RPCCall*. Esta página modela o comportamento

de uma chamada remota, assumindo que a serialização de parâmetros já foi efetuada. Um *user thread* no lugar *P1* (ficha) indica que ele está pronto para iniciar a chamada remota, habilitando a transição *Send*. A guarda desta transição garante que o pacote será inserido no buffer do *shell* que contém o *user thread* que iniciou a chamada remota. Ao enviar o pacote, o *user thread* vai para o lugar *Blocked*, onde permanecerá até receber o retorno, através da adição do pacote enviado no lugar *OutBuf*. Quando isto acontecer a transição *Recv* é habilitada e o *user thread* poderá receber a resposta. Ao receber a resposta três situações podem ocorrer:

1. O retorno foi Ok: uma ficha vai ser depositada no lugar *No_error*, que representa a situação na página *Send* em que a resposta está pronta para ser desserializada;
2. O retorno contém um erro: duas situações podem ocorrer:
   - O *user thread* volta para o estado inicial para refazer a chamada remota;
   - O *user thread* vai para o lugar *Error* que representa a situação na página *Send* em que o retorno do RPCCall foi um erro.

Estas situações são avaliadas pelas funções *Timeout*, *connBrocken* e *retry*. Os valores de erros são setados na página *Network* quando ocorrer a transição *Network_error*.

Página Get

Na Figura B.9 é apresentada a página *Get*. Esta página modela a situação em que o RPCCall ocorreu normalmente e a página *Send* está recebendo um retorno. A página *Get* é responsável por desserializar o resultado. Diferente da serialização (página *Put*) a desserialização foi modelada em uma única transição *Unserialize*. Após a desserialização, a tabela local de OID's é consultada para verificar se existe ou não o OID do objeto resultante. Os *monitores* são utilizados para controlar o acesso à tabela (lugar *LShellMonitor*). As transições *lObj_not_found* e *lObj_found* indicam se o OID foi encontrado localmente ou não. Se o OID do objeto foi encontrado na tabela de OID's locais, o processo de desserialização é concluído. Caso o OID não for encontrado na tabela de OID's locais, o OID será pesquisado na tabela que contém os OID's remotos (transição *RemObj_findByOID*). Se o OID é encontrado na tabela de OID's remotos, o processo é finalizado. Caso contrário, o OID do objeto

deve ser inserido na tabela de OID's remotos (transição *RemObj_insert*). Neste momento os *monitores* também são usados.

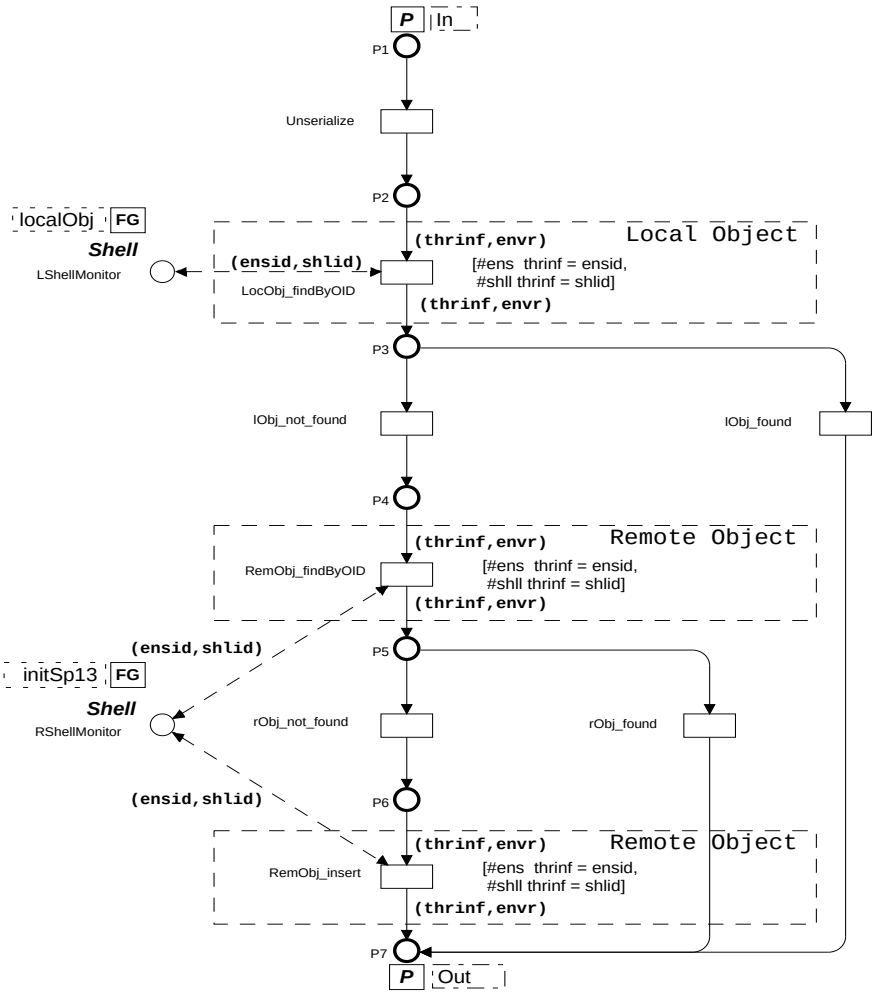


Figura B.9: Página *Get*

B.6 Parte Receiver

Esta parte modela o recebimento de uma requisição para uma chamada de objeto ou um novo OID. A recepção de uma mensagem pode acontecer de duas formas:

1. Se a mensagem é uma requisição ao *shell*, ela é tratada na página *Listen*.

2. Se a mensagem é uma requisição de um OID para um *ensemble*, ela é tratada na página *OIDGen*.

Página Listen

Na Figura B.10 é apresentada a página *Listen*. Esta página modela o recebimento das requisições no *shell*. No lugar *P1* existe um *listener thread* para cada *shell*. Se existir uma mensagem no lugar *OutBuf* direcionada para o *shell* e o *listener thread* do *shell* estiver disponível, a transição *Listen* poderá ocorrer. Quando a transição ocorrer, o pacote será retirado do lugar *OutBuf* e o emissor será registrado para que o resultado possa ser enviado de volta (função *AssignReplyPar*). O *listener thread* alocará um *worker thread* (sob a proteção de monitores – lugar *WorkerPoolMonitor*) e voltará para o seu estado inicial (transição *Fork_and_GetWorker*). Cada *Shell* tem um número ilimitado de *workers threads* disponíveis. Neste estado o trabalho do *listener thread* foi concluído e o restante será feito pelo *worker thread*. Os detalhes do trabalho do *worker thread* são modelados na página *Execute*. Quando o trabalho é finalizado, o resultado é enviado de volta para a rede (transição *Release_and_Reply*).

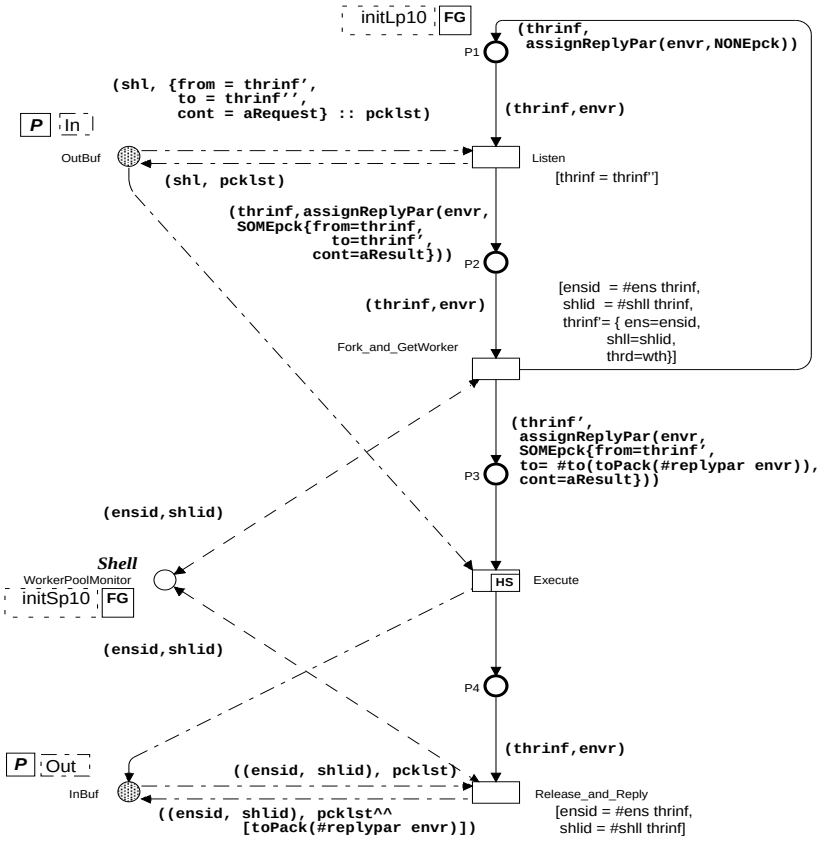


Figura B.10: Página *Listen*

Página Execute

Na Figura B.11 é apresentada a página *Execute*. Esta página modela as ações básicas executadas pelo *worker thread*. Primeiramente os parâmetros das mensagens serão desserializados transição de substituição *Get*. Em seguida, o trabalho efetivo é executado *transição Method_Call*. Por fim, o resultado é serializado, estando pronto para ser retornado ao solicitante transição de substituição *Put*.

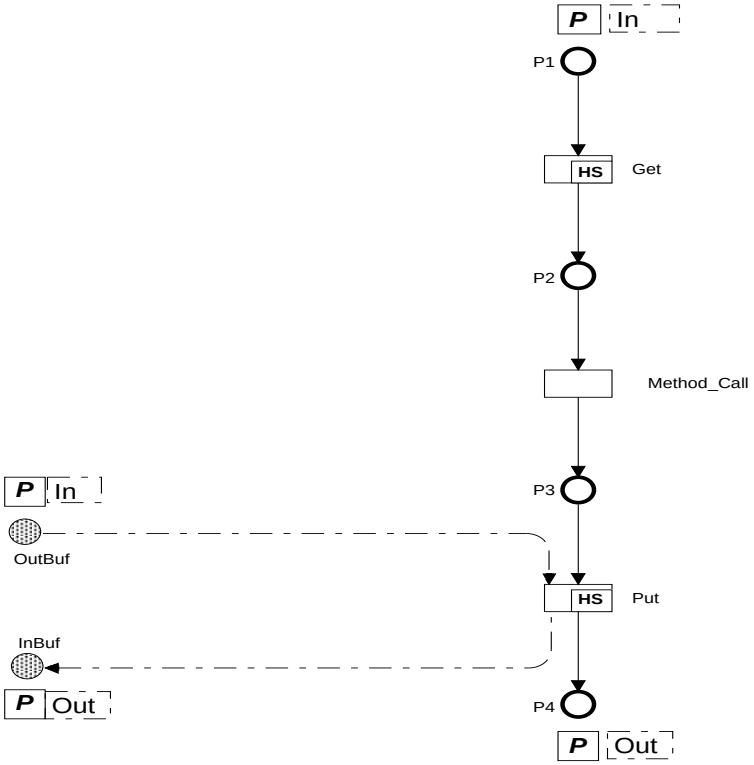


Figura B.11: Página *Execute*

Página OIDGen

Na Figura B.12 é apresentada a página *OIDGen*. Esta página modela a geração de OID's. Inicialmente existe um *listener thread* para cada *ensemble* no lugar *P1*. Se existir uma mensagem no lugar *OutBuf* direcionada para o *ensemble* e o *listener thread* estiver disponível, a transição *Listen* poderá ocorrer. Quando esta transição ocorrer, o pacote será retirado do lugar *OutBuf* e o emissor será registrado, para que o resultado possa ser enviado de volta (função *AssignRPCPar*). Em seguida o OID é gerado transição *Generate_OID*. Ao ocorrer a *transição Reply* o *listener thread* voltará para o estado inicial e o resultado será enviado de

volta para a rede.

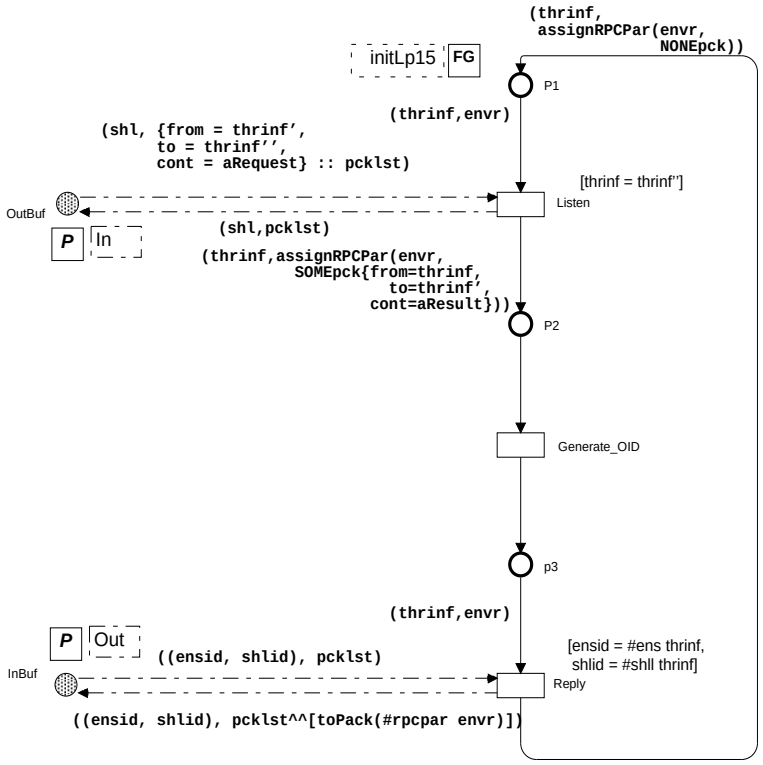


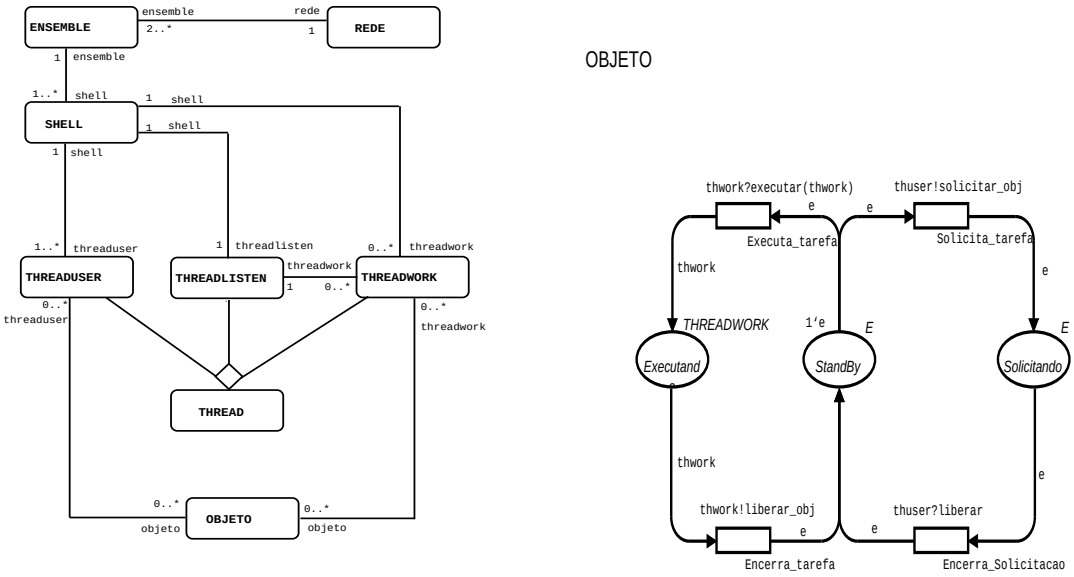
Figura B.12: Página *OIDGen*

B.7 Diagrama de Classes e Classe OBJETO

B.8 Classe THREADUSER

Na Figura B.14 é apresentado o corpo da classe *THREADUSER*. Inicialmente, apenas a transição *Inicia_tarefa* está habilitada, indicando que o *thread user* recebeu a mensagem *objeto?solicitar_obj* de alguma instância da classe *OBJETO*. Com o disparo desta transição a ficha do lugar *StandBy* e removida e a referência ao objeto que enviou a mensagem é armazenada no lugar Iniciando. Todos os lugares desta rede, com excessão do lugar *StandBy*, armazenam fichas do tipo *Objeto*. Isto acontece porque depois que o *user thread* sai do estado *standBy*, é preciso armazenar o objeto que fez a solicitação para que no final o *thread user* possa retornar a informação ao objeto correto.

A partir do estado *Iniciando*, a transição *Recupera_recurso* poderá ocorrer. Quando esta transição ocorrer a mensagem *shell!in_monitor(recurso)* será enviada para a classe *SHELL*,



(a) Diagrama de Classes

(b) Classe OBJETO

Figura B.13: Diagrama de Classes e Classe OBJETO

pois um recurso precisa ser alocado exclusivamente. O tipo de monitor que o *user thread* deseja acessar é enviado como parâmetro, já que os monitores de outros recursos não são afetados. A transição *Recurso_ok* irá ocorrer quando o *user thread* receber a mensagem *shell?out_monitor* do *shell*. As transições *Inicia_serializacao* e *Finaliza_serializacao*, modelam a troca de mensagens com o próprio *user thread* para iniciar e encerrar a serialização de parâmetros.

Após a serialização da mensagem, uma ficha será depositada no lugar *Pronto* e a mensagem *shell!passar_mens(iniciaMens(req))* é enviada para o *shell* a partir do disparo da transição *Envia_mensagem*. Assim, uma ficha é adicionada no lugar *MRetornada* e o *user thread* vai esperar pelo retorno do resultado. Ocorrendo a transição *Ok*, a mensagem recebida será desserializada. Se houver erro a desserialização não será efetuada.

A transição *Libera_recurso* representa mais um acesso ao monitor. Quando a mensagem *shell?out_monitor* é recebida pelo *user thread* através da transição *Encerra_tarefa* o *user thread* enviará a mensagem *objeto!liberar* para o objeto que fez a solicitação e uma ficha será adicionada no lugar *StandBy*, deixando o *user thread* disponível para receber novas

solicitações dos objetos.

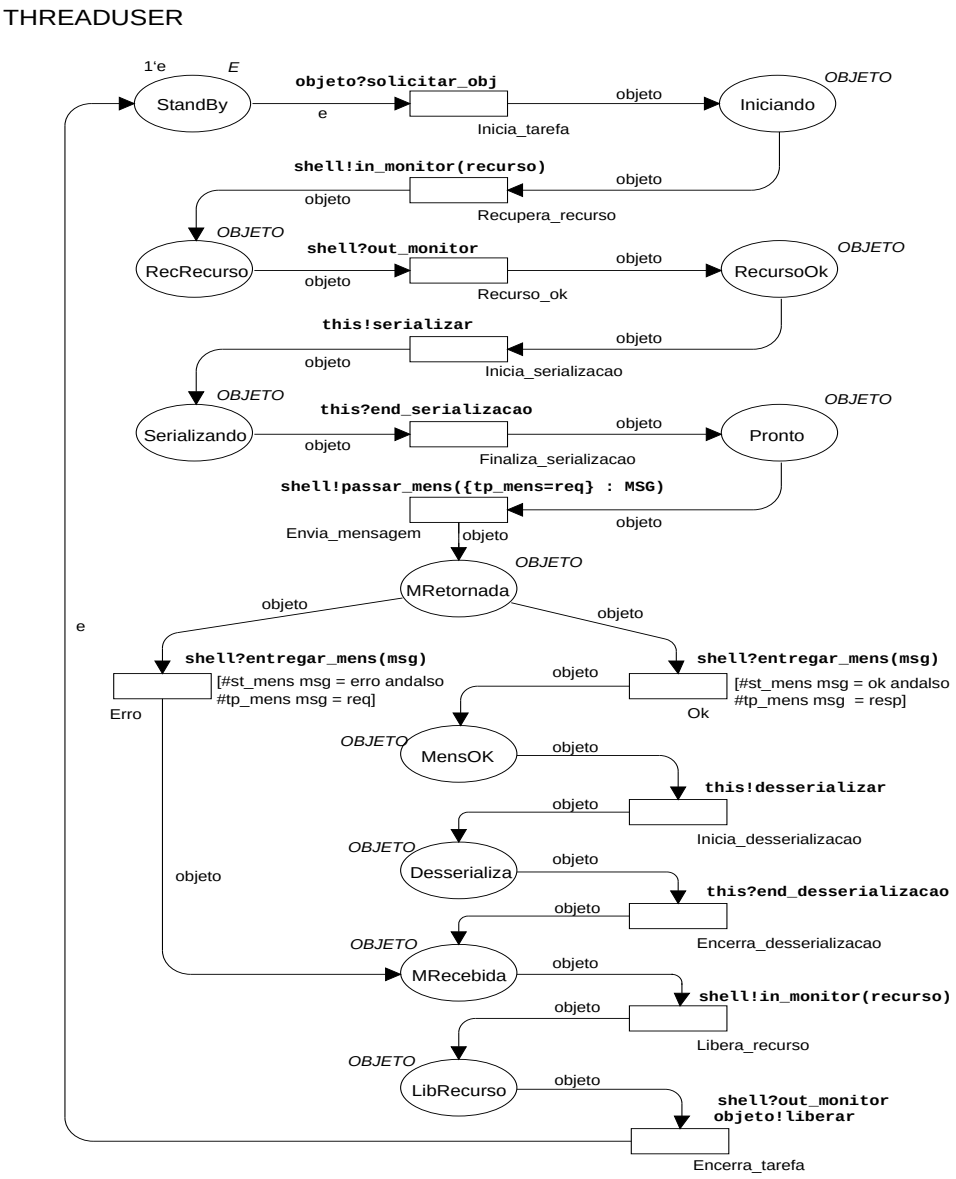


Figura B.14: Classe THREADUSER

B.9 Classe THREADLISTEN

Na Figura B.15 é apresentado o corpo da classe *THREADLISTEN*. Esta rede é composta por três lugares (*StandBy*, *CriandoTW* e *EndCriaTW*). Inicialmente há uma ficha no lugar *StandBy*, indicando que o *listener thread* está disponível. Quando o *listener thread* receber a mensagem *shell?entregar_mens(mens)* do *shell*, uma ficha será depositada no lugar *CriandoTW*. A partir deste estado a transição *Cria_thread_work* poderá ocorrer, a qual representa

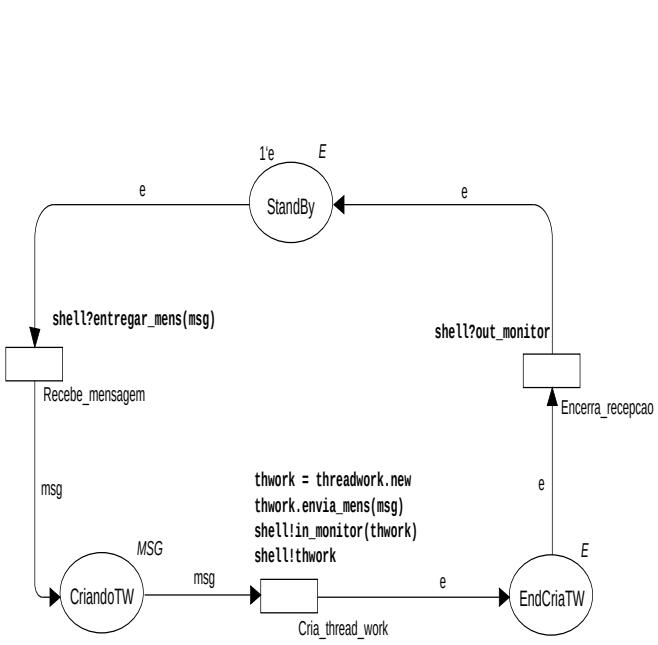


Figura B.15: Classe THREADLISTEN

a criação de um *worker thread* pelo *listener thread* (mensagem *threadwork.new(msg)*. Ao mesmo tempo o *listener thread* entrará no monitor porque a criação de *workes threads* também é controlada por um monitor (mensagem *shell!in_monitor(thwork)*). A transição *Encerra_recepcao* irá ocorrer quando o *listener thread* receber a mensagem *shell?out_monitor* do *shell* e o *listener thread* voltará para o estado *StandBy*.

B.10 Classe THREADWORK

Na Figura B.16 é apresentado o corpo da classe *THREADWORK*. A transição *Start* é disparada quando o *worker thread* receber a mensagem *tl?new (msg)* do *listener thread*. A partir do estado *Inicializacao* o *worker thread* vai solicitar a desserialização da mensagem recebida e aguardar o fim da desserialização através das transições *Inicia_desserializacao* e *Encerra_desserializacao* (mensagens *this!desserializar* e *this?end_desserialzacao*). Após a desserialização da mensagem um dos objetos vai ser selecionado para executar a tarefa solicitada e o *worker thread* vai aguardar o fim da execução da tarefa pelo objeto, através do disparo das transições *Chama_Obj* e *Encerra_chamada_obj*. Existindo uma ficha no lugar *Tarefa_Ok* as transições *Inicia_serializacao* e *Finaliza_serializacao* podem ocorrer. Para re-

tornar a mensagem, a transição *Envia_mensagem* será disparada e o *worker thread* acessará o monitor através da mensagem *shell!in_monitor(thwork* e enviará a mensagem de volta, entregando-á ao *shell* (mensagem *shell.passar_mens (setMens(msg, resp)*). A transição *Finaliza_thread* irá disparar quando o monitor liberar o *worker thread*, através do recebimento da mensagem *shell?out_monitor*. Ao receber esta mensagem o *worker thread* será destruído, mensagem *this.destoy*.

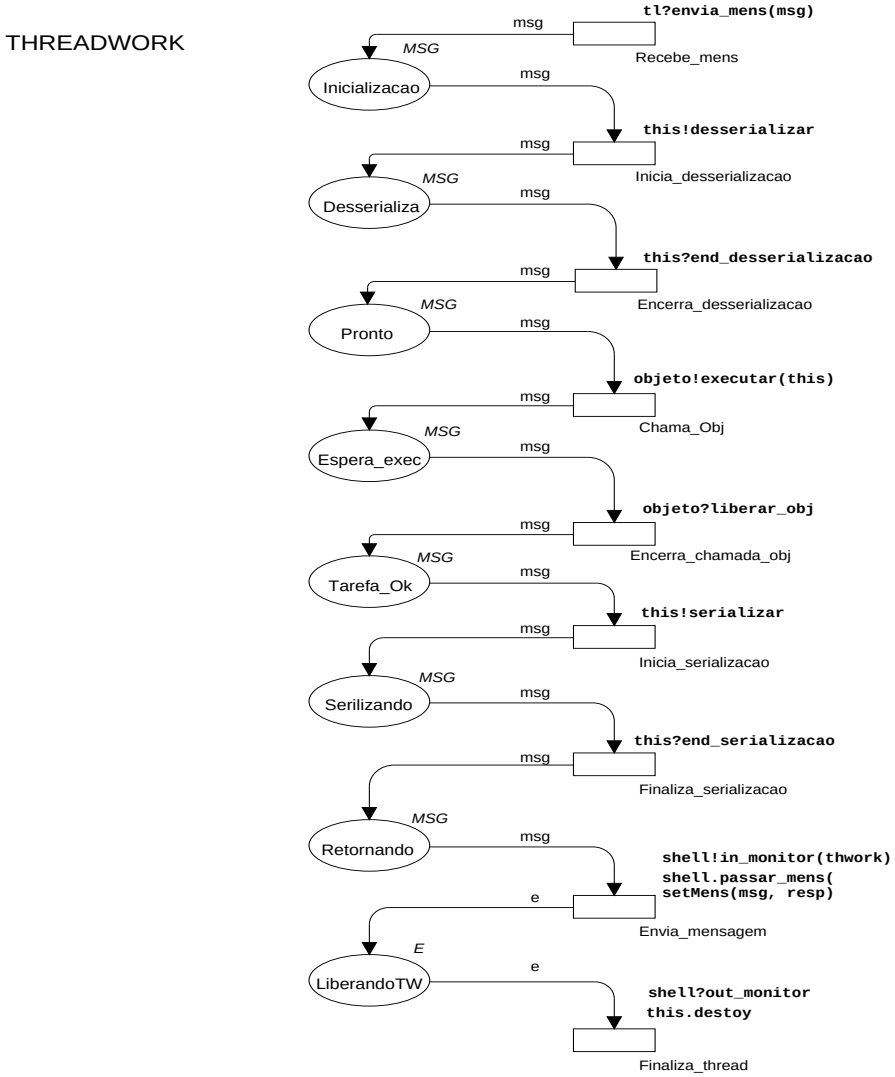


Figura B.16: Classe THREADWORK

B.11 Classe THREAD_Serializacao

Na Figura B.17 é apresentado o corpo da classe *THREAD_Serializacao*. A transição *Inicia* irá ocorrer quando a mensagem *this?serializar* for recebida. A recepção é através da variável

this porque a classe *THREAD_Serializacao* faz parte do *thread*, assim ele recebe a mensagem do próprio *thread* a que ela pertence.

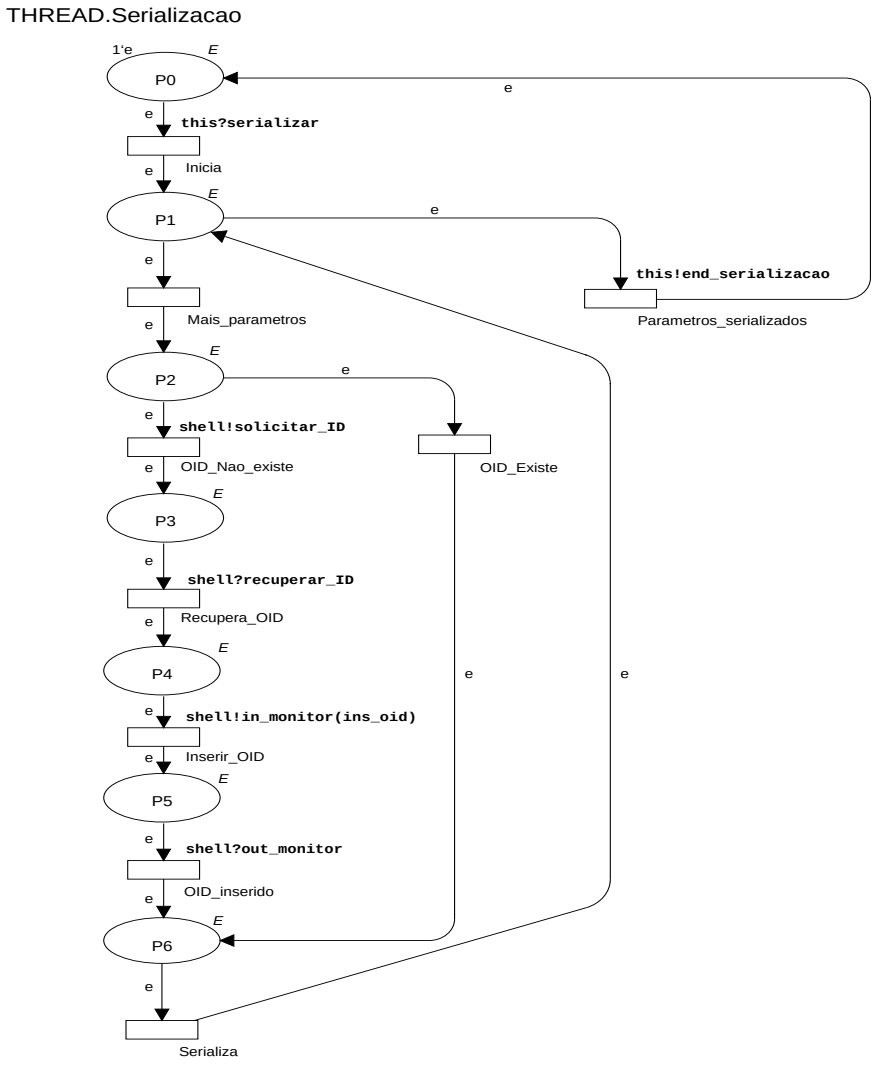


Figura B.17: Classe THREAD_Serializacao – Serialização de Parâmetros

No estado *P1* duas situações podem ocorrer. Se não existir mais parâmetros, o processo é encerrado e a transição *Parâmetros_serializados* é disparada, enviando a mensagem *this!end_serializacao* ao *user thread* que está esperando o final da serialização. Se existir parâmetros a serem serializados, a transição *Mais_parametos* é disparada e uma ficha é adiconada no lugar *P2*. A partir deste estado pode acontecer duas situações: existe a informação do identificador do objeto cujo o parâmetro deve ser serializado localmente, ou esta informação deve ser solicitada remotamente. Se a informação existe, a transição *OID_Existe* é disparada e uma ficha será depositada no lugar *P6*. Se a informação não existir, deverá ser

solicitada ao *shell* através da mensagem *shell!Solicitar_ID*. Quando o *shell* retorna a informação pelo disparo da transição *Recupera_OID*, uma ficha é colocada no lugar *P4* e será feito o acesso ao monitor para que o *user thread* faça a inserção do ID na tabela local e assim não seja necessário nova solicitação ao *shell*. Quando este processo terminar, uma ficha estará no lugar *P6* e a transição *Serializa* poderá ocorrer. Nesta transição é que está representado efetivamente a serialização dos parâmetros. Depois da serialização a ficha é colocada no lugar *P1* e o ciclo se repete até que não existam mais parâmetros a serem serializados.

B.12 Classe THREAD_Desserializacao

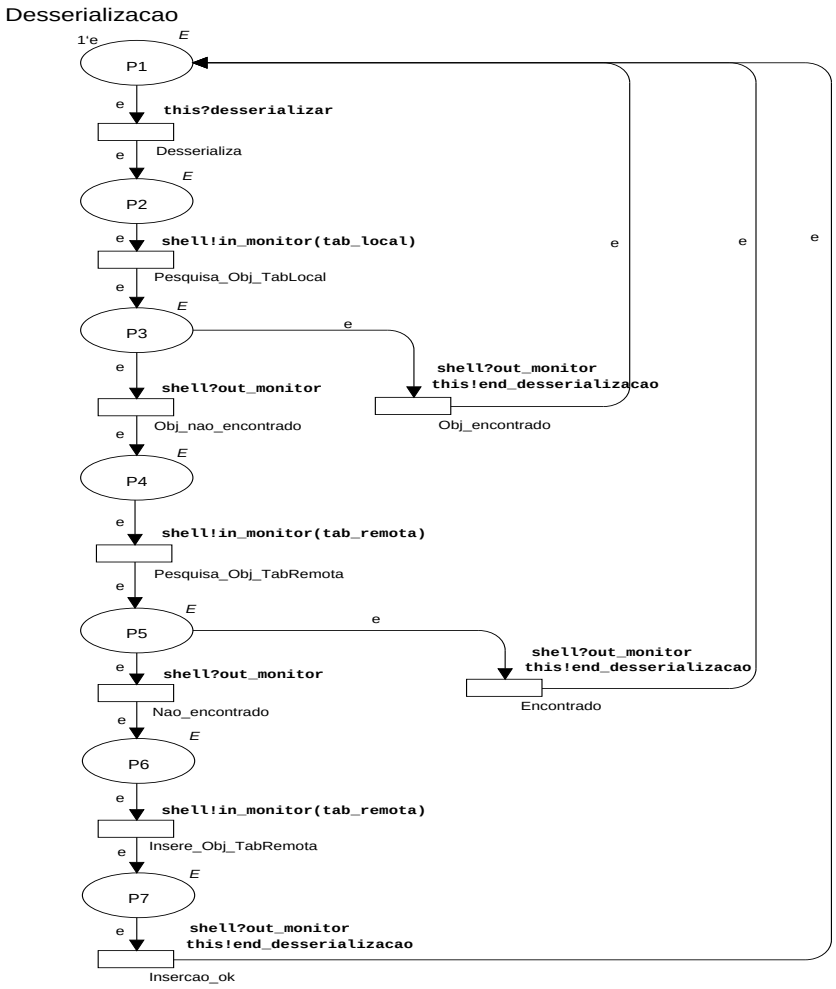


Figura B.18: Classe THREAD_Desserializacao – Desserialização de Parâmetros

Na Figura B.18 é apresentado o corpo da classe *THREAD_Desserializacao*. O corpo desta rede representa a desserialização de mensagens. Inicialmente apenas a transição *Des-*

serializa está habilitada (recebimento da mensagem *this?desserializar*). Depois que a mensagem é desserializada os valores encontrados são pesquisados nas tabelas do *shell*. Isto é representado pela transição *Pesquisa_Obj_TabLocal* e como o acesso a esta tabela é compartilhado, a mensagem *shell!in_monitor(tab_local)* é enviada ao *shell* para que seja feito o controle de acesso, ou seja, a transição só poderá ser disparada quando o monitor estiver disponível. Quando a transição é disparada, uma ficha é adicionada no lugar *P3*. Se o objeto já existe na tabela local, a transição *Obj_encontrado* é disparada ao receber a mensagem *shell?out_monitor* do *shell* e a mensagem *this!end_desserializacao* é enviada ao *user thread* indicando o final da desserialização solicitada. Caso o objeto não seja encontrado na tabela local, a transição *Obj_nao_encontrado* é disparada ao receber a mensagem *shell?out_monitor* e uma ficha será depositada no lugar *P4*.

A partir do estado *P4*, será feito uma pesquisa na tabela de IDs remotos (transição *Pesquisa_Obj_TabRemota*). O acesso a esta tabela também é controlado, a mensagem *shell!in_monitor(tab_remota)* é enviada ao monitor do *shell*.

No estado *P5* as transições *Encontrado* e *Nao_encontrado* podem ocorrer. Se o ID do objeto não é encontrado na tabela remota, uma ficha será depositada no lugar *P6* e a transição *Insere_Obj_TabRemota* será disparada, representando a inserção do ID não encontrado na tabela remota. Ocorrendo a transição *Insercao_ok* a mensagem *this!end_desserializacao* é enviada para o *thread* solicitante informando o final da desserialização e o monitor será liberado, através da mensagem *shell?out_monitor*.

B.13 Classe SHELL_monitores

Na Figura B.19 é apresentado o corpo da classe *SHELL_Monitores*. Esta rede representa os monitores que controlam os diversos recursos compartilhados no sistema. No lugar *StandBy* existe uma ficha para cada tipo de monitor. Quando o *shell* receber a mensagem *th?in_monitor(monitor)* de algum *thread*, a transição *Entra_monitor* é disparada. O tipo do monitor é enviado como parâmetro pelo *thread* que solicita o acesso ao monitor. A partir do estado *Monitorando* a transição *Libera_monitor* pode ser disparada e o *thread* solicitante receberá a mensagem *th?out_monitor*. Após, o recurso é liberado, tornando-se disponível novamente.

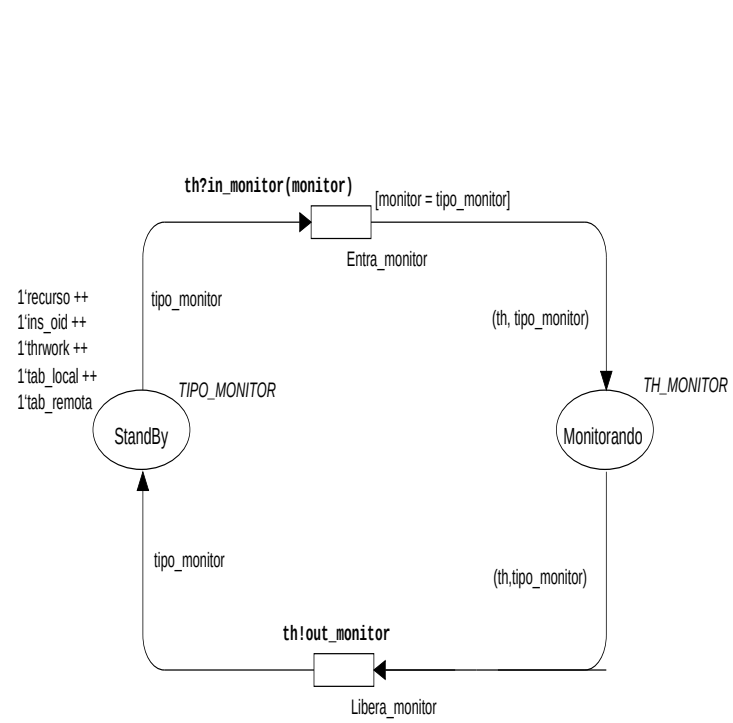


Figura B.19: Classe SHELL_monitores – Monitores

B.14 Classe SHELL_InformarID

Na Figura B.20 é apresentado o corpo da classe *SHELL_InformaID*. Esta rede é acionada quando um *user thread* precisa recuperar um ID de um objeto para o qual ele deseja enviar uma solicitação. Nesta página existe também um monitor (lugar *Monitor_ID*). Este é o único monitor que não está no corpo da classe *Shell_monitores*, porque quando um *user thread* deseja solicitar um ID a um *shell*, nenhum outro *user thread* pode fazer esta solicitação.

Quando a transição *Solicita_ID* ocorrer, o *user thread* será armazenado no lugar *SolicitandoID* para que seja possível ao *shell* retornar a informação para o *user thread* correto. Neste estado duas situações podem ocorrer: o *shell* contém a informação do ID ou o *shell* tem que buscar o ID remotamente. Se a transição *Chace_ID* disparar, a mensagem *th!recuperar_ID* será enviada ao *user thread* solicitante e uma ficha será depositada no lugar *Monitor_ID*, liberando o *shell* novamente para atender novas requisições de solicitação de ID. Ocorrendo a transição *ID_Remoto*, uma ficha *thread* será depositada no lugar *SolicitandoID_Global*. Neste estado, apenas a transição *Solicita_ID_Ensemble* poderá ocorrer e quando isso acontecer, a mensagem *ensemble.entregar(iniciaMens(ID))* será enviada ao

ensemble, indicando que a solicitação do ID terá que ser feita a um *ensemble* (que pode ser remoto).

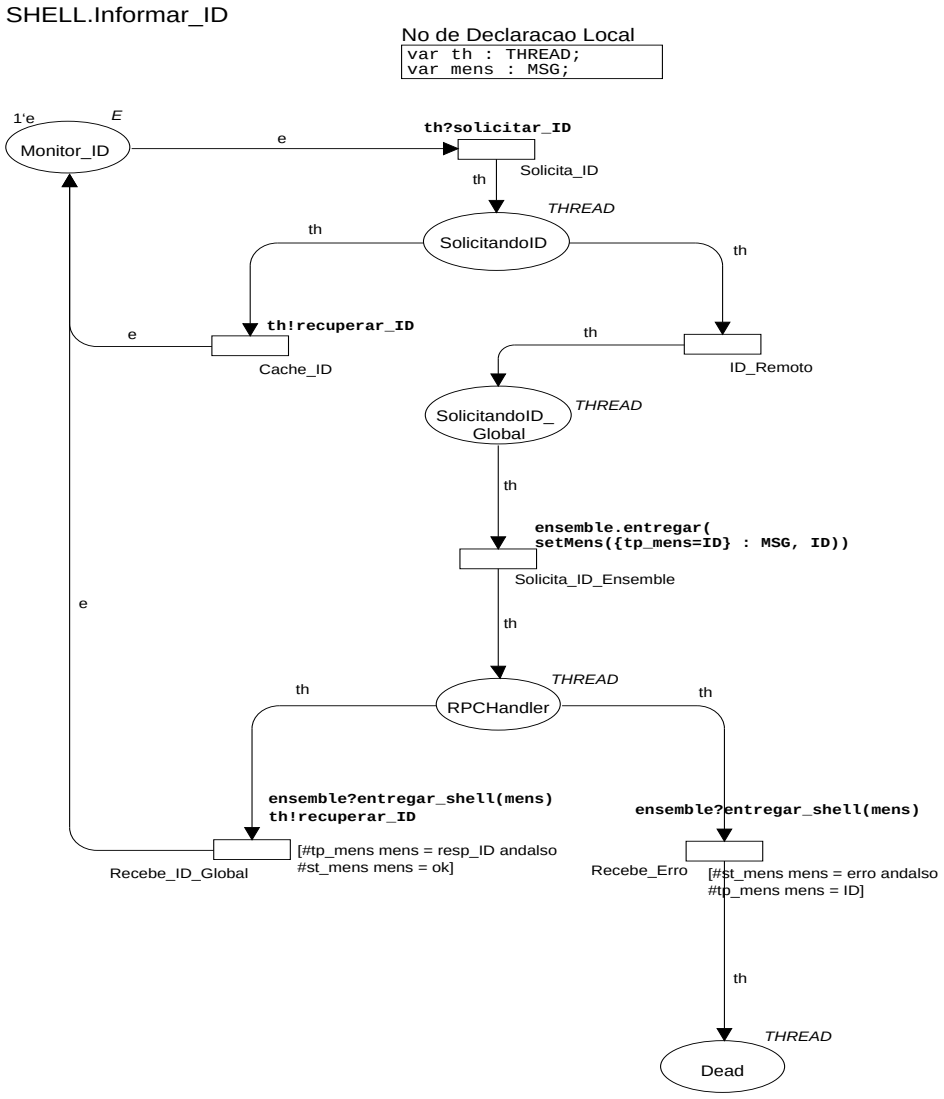


Figura B.20: Classe SHELL_InformarID – Informar ID

No estado *RPCHandler* as transições *Recebe_ID_Global* e *Recebe_Erro* podem ocorrer. Se a informação foi retornada normalmente a transição *Recebe_ID_Global* será disparada e a mensagem *th!recuperar_ID* será enviada ao *user thread* que solicitou a informação e uma ficha é depositada no lugar *Monitor_ID*, liberando o *shell*. Se a mensagem contém algum erro, a transição *Recebe_Erro* é disparada e o *user thread* vai para o lugar *Dead*, não podendo mais ser utilizado.

B.15 Classe SHELL_RPC

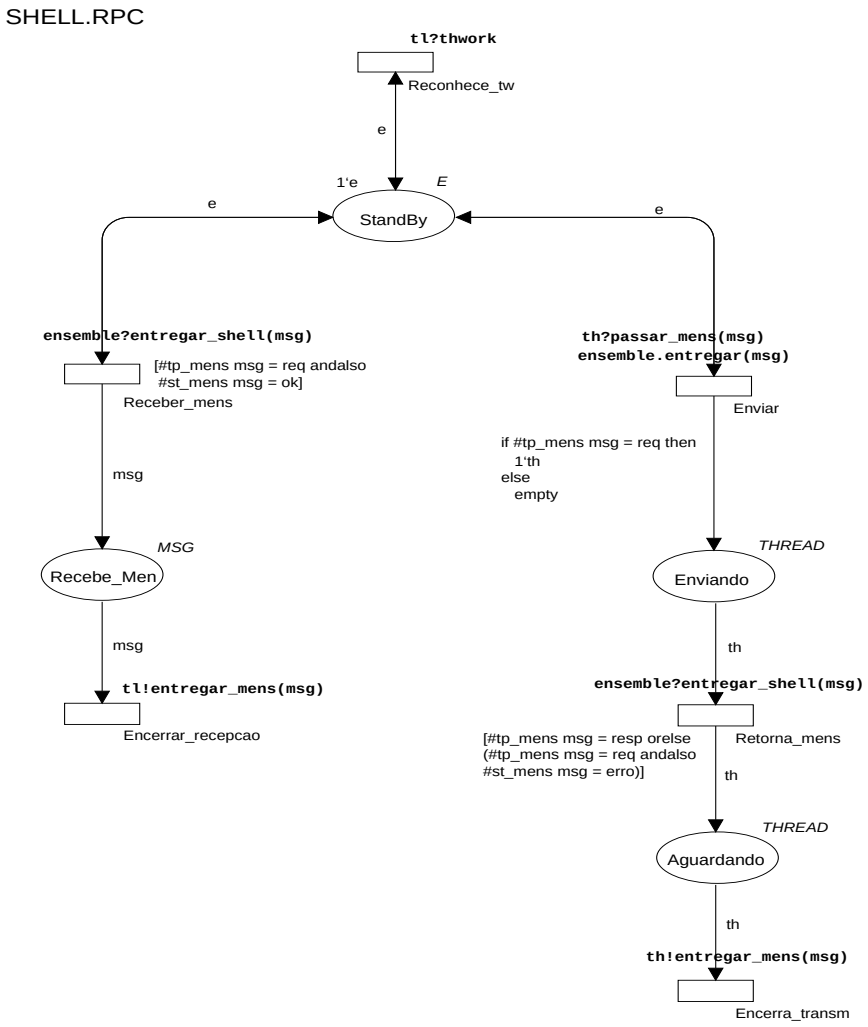


Figura B.21: Classe SHELL_RPC

Na Figura B.21 é apresentado o corpo da classe *SHELL_RPC*. Esta rede representa uma das funcionalidades da classe *SHELL*, que é a entrega e recepção de uma mensagem que deve ser enviada pela, ou que chegou da rede. Inicialmente, há uma ficha no lugar *StandBy* indicando que o *shell* está pronto para enviar ou receber uma mensagem. O lado direito da figura representa a transmissão de uma mensagem. A transição *Enviar* será disparada quando o *shell* receber a mensagem *th?passar_mens(mens)*. A mensagem recebida pelo *shell* (através da variável *mens*) é repassada para o *ensemble* através da mensagem *ensemble.entregar(mens)*. Se a mensagem a ser enviada for uma requisição, o *thread* será armazenado no lugar *Enviando* para que o retorno possa ser entregue ao *thread* que solicitou a

transmissão. Se não for uma requisição é porque é uma resposta vinda de um *thread*. Assim, o *shell* não precisa armazenar nenhuma informação porque não vai precisar devolver a mensagem. A partir do estado *Enviando* a transição *Retorna_mens* poderá ocorrer quando receber a mensagem *ensemble?entregar_shell(mens)* do *ensemble*. Esta mesma situação acontece no lado direito da figura, com a transição *Receber_mens*. A mensagem recebida para disparar esta transição é a mesma, ou seja, *ensemble?entregar_shell(mens)*. Isso porque, se for um retorno de uma solicitação feita, a mensagem deve ser entregue ao *thread* solicitante, mas se for a recepção de uma solicitação (transição *Receber_mens*), o *shell* irá simplesmente repassar a informação ao *listener thread* através da mensagem *threadlisten!entregar_mens(msg)* (transição *Encerrar_recepcao*). Se for um retorno de uma solicitação feita, a transição *Encerra_trasm* será disparada e a mensagem *th!entregar_mens(msg)* será retornada ao *thread* solicitante.

B.16 Classe ENSEMBLE

Na Figura B.22 é apresentado o corpo da classe *ENSEMBLE*. A ficha do lugar *StandBy* indica que o *ensemble* está pronto para transmitir ou receber uma mensagem. A parte superior da figura representa a situação em que o *ensemble* vai enviar uma mensagem para a rede. O envio da mensagem poderá acontecer em uma das três transições *Envia_mens, Envia_resp* e *Reenvia_mens*. As guardas associadas às transições e o componente que está enviando a mensagem garantem o disparo da transição correta. Se o *ensemble* receber a mensagem *shell?entregar(mens)* de um *shell*, pode ser que o *shell* esteja, transmitindo ou retornando uma mensagem. O tratamento é diferenciado devido a restrição de não serem considerados erros no retorno de uma transmissão e pelo fato de existirem funções diferentes para inserção dos campos origem e destino na mensagem (se for uma transmissão, um *ensemble* destino será escolhido pela rede, se for um retorno o *ensemble* origem passa a ser destino e vice-versa). A transmissão de uma mensagem vai fazer com que a transição *Envia_mens* seja disparada e a mensagem *rede.send(setshell (mens, shell))* será passada para a rede. Se for a transmissão de uma resposta, a transição *Envia_resp* será disparada e a mensagem *rede!reply(setMens(mens, resp))* será entregue a rede. A transição *Reenvia_resp* será disparada quando a mensagem *this?entregar(mens)* for recebida pelo próprio *ensemble*. Esse é o caso de haver ocorrido um erro na transmissão e a mensagem ser retransmitida. O disparo de

qualquer uma destas transições vai retirar uma ficha do lugar *StandBy* e colocar uma ficha no lugar *Env_Mens*. A transmissão será considerada encerrada quando a mensagem *rede?tratar* for recebida da rede e a transição *Encerra_envio* for disparada, tornando o *ensemble* disponível novamente.

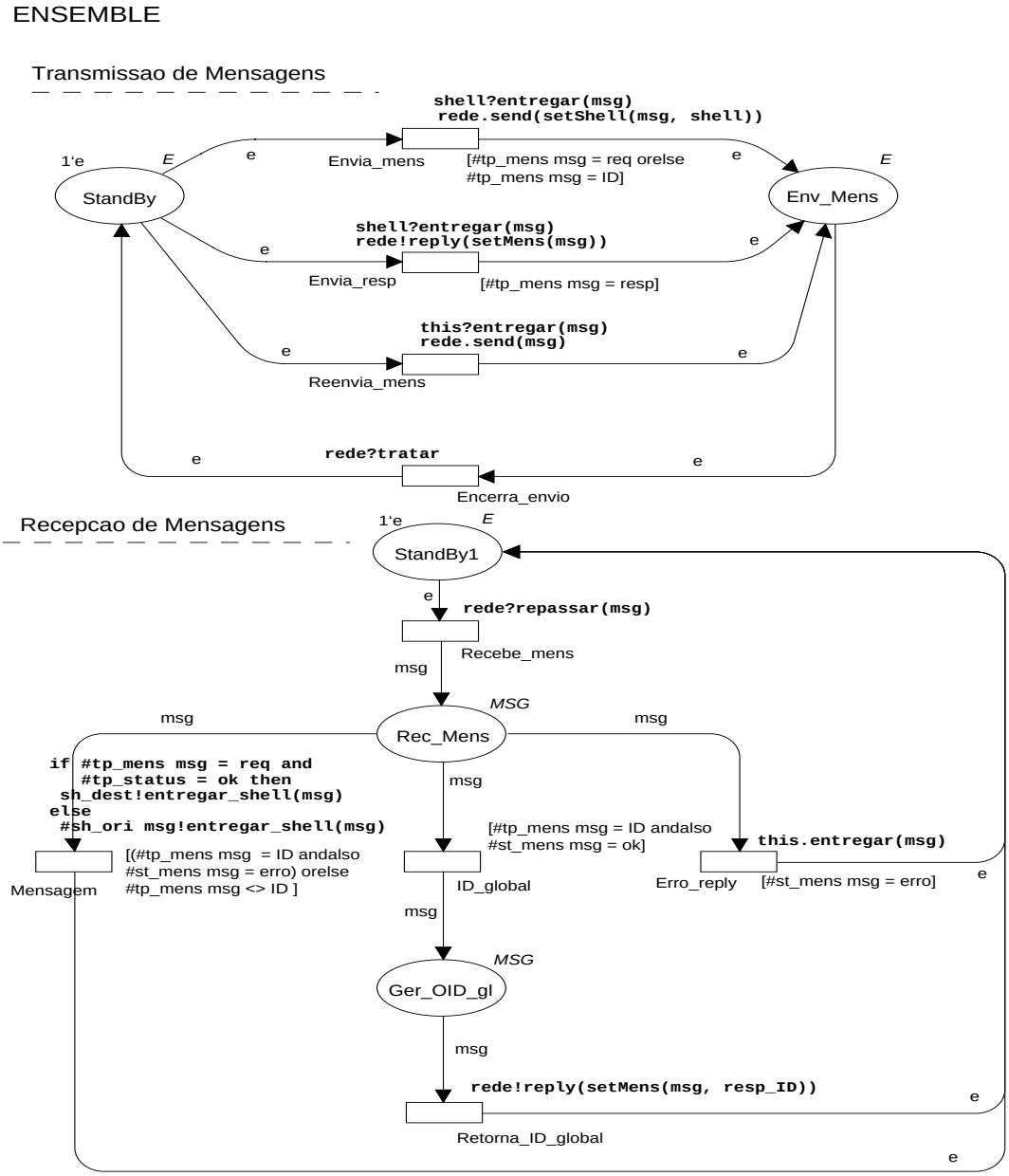


Figura B.22: Classe ENSEMBLE

A parte inferior da página, que representa a recepção de uma mensagem por um *ensemble*, acontece quando o *ensemble* está em *StandBy* e recebe a mensagem *rede?repassar(mens)*

da rede (transição *Recebe_mens*). A mensagem recebida é armazenada no lugar *Rec_Mens*, indicando que o *ensemble* está recebendo determinada mensagem naquele instante e não está mais disponível até que a recepção seja encerrada. A partir deste estado três situações podem ocorrer. Se houver algum erro de transmissão, a mensagem será retornada da rede para o *ensemble* sem ser entregue ao destino e duas transições podem ser disparadas *Mensagem* ou *Erro_reply*. Se a transição *Mensagem* ocorrer, a mensagem *sh_ori msg!entregar_shell(msg)* será retornada ao *shell*. Se a transição *Erro_Reply* ocorrer, a mensagem será retransmitida através da mensagem *this.entregar(msg)*.

Se o *ensemble* está recebendo uma mensagem sem erro que não é uma solicitação de ID, a transição *Mensagem* poderá ocorrer. Se o *ensemble* estiver recebendo uma solicitação, ele vai selecionar um dos seus *shells* para entregar a mensagem *sh_dest!entregar_shell (msg)*. Se o *ensemble* estiver recebendo um retorno, ele vai entregar ao *shell* que fez a solicitação, mensagem *sh_ori msg!entregar_shell (msg)*.

Caso a mensagem recebida seja uma solicitação de ID, o próprio *ensemble* executará a tarefa de localizar o ID e retorná-lo, porque ele mantém uma tabela de ID's globais. Quando a transição *ID_global* ocorrer, uma ficha será adicionada no lugar *Ger_OID_gl* e a transição habilitada passa a ser *Retorna_ID_global*, que transmiste a mensagem *rede!reply(setMens(Msg, Resp_ID))* para a rede informando o ID. Depois do disparo desta transição, o *ensemble* voltará para o estado *StandBy*.

B.17 Classe REDE

Na Figura B.23 é apresentado o corpo da classe *REDE*. O corpo desta classe possui três lugares (*Disponivel, Trantando_mens* e *Ocupada*). A partir do estado *Disponível* duas transições podem ocorrer dependendo da mensagem que a rede receber. Uma mensagem que está sendo transmitida de um *ensemble* a outro chega na rede através da mensagem *ens_?send(msg)* e a transição *Recebe_mens* é disparada. O *ensemble* de origem e destino são armazenados na mensagem, se for a primeira vez que a mensagem está sendo transmitida. Caso contrário, a mensagem está sendo retransmitida em função de algum erro ocorrido e os valores já estão armazenados. O disparo da transição *Recebe_mens* vai fazer com que a mensagem seja armazenada no lugar *Trantando_mens* e novamente duas situações podem acontecer. Pode ser inserido um erro na transmissão (transição *Insere_erro*), ou a mensagem pode ser recebida

pela rede normalmente (transição *Mens_ok*). Qualquer uma das duas transições irá enviar a mensagem *ens_ori msg!tratar* para o *ensemble* que enviou a mensagem para liberá-lo de sua transmissão, e vai inserir a mensagem depois de ter alterado o status a depender da transição que tenha sido disparada. Dessa forma, a mensagem será depositada no lugar *Ocupada*, indicando que a rede possui uma mensagem para ser entregue a algum *ensemble* ligado a ela. Quando a transição *Transmite_mens* disparar, a mensagem *ens_dest msg.repassar(msg)* será entregue ao *ensemble* destino. Caso a mensagem tenha algum erro, ela será devolvida ao *ensemble* de origem através da mensagem *ens_ori msg.repassar(msg)*. Após o disparo desta transição a rede voltará para o estado *Disponível*.

REDE

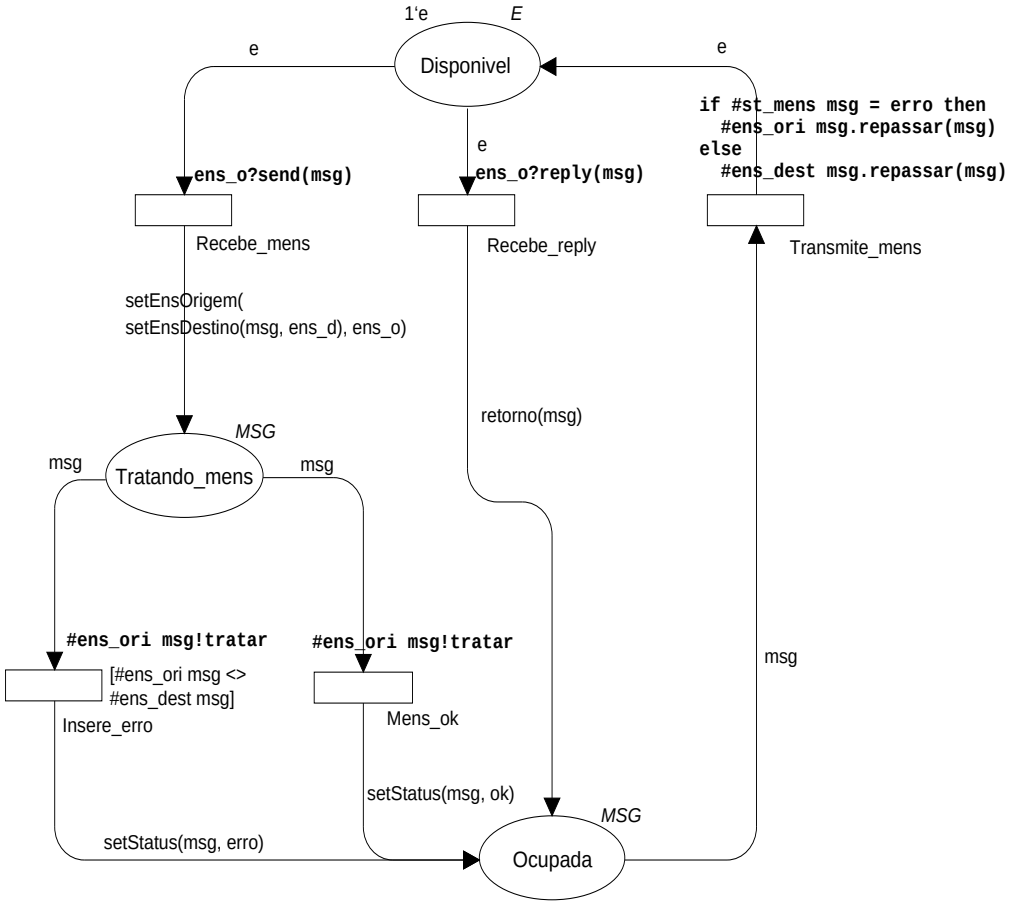


Figura B.23: Classe REDE

Apêndice C

Terceiro Experimento de Modelagem

C Protocolo *OSPF* – HCPN e RPOO

Os produtos de modelagem do protocolo *OSPF* usando HCPN, foram desenvolvidos por uma aluna [Car02] do curso de mestrado do Departamento de Ciência da Computação desta Universidade.

C.1 Página de Hierarquia

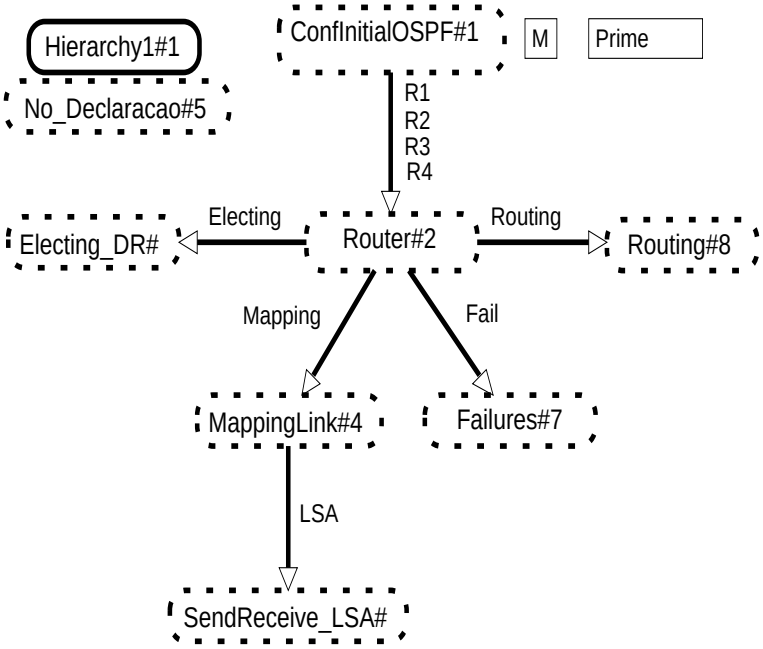


Figura C.1: Página de *Hierarquia*

Na Figura C.1 é apresentada a página de *hierarquia* do modelo, que é constituído de oito páginas. A página *ConfInitialOSPF* é utilizada para a inicialização dos roteadores do modelo. As inscrições *R1, R2, R3* e *R4* representam os nomes das transições de substituição presentes nesta página, as quais representam uma abstração dos quatro roteadores instanciados no modelo.

As páginas *Electing_DR, Routing* e *Failures* modelam a escolha do roteador designado, roteamento de pacotes de dados, falha de um roteador, respectivamente. A página *MappingLink* modela o mapeamento de um enlace, pois dois roteadores comunicam-se através de um enlace. Finalmente, a página *SendReceive_LSA* modela a troca de pacotes de anúncio de estado de enlace (LSA) entre os roteadores.

C.2 Página ConfInitialOSPF

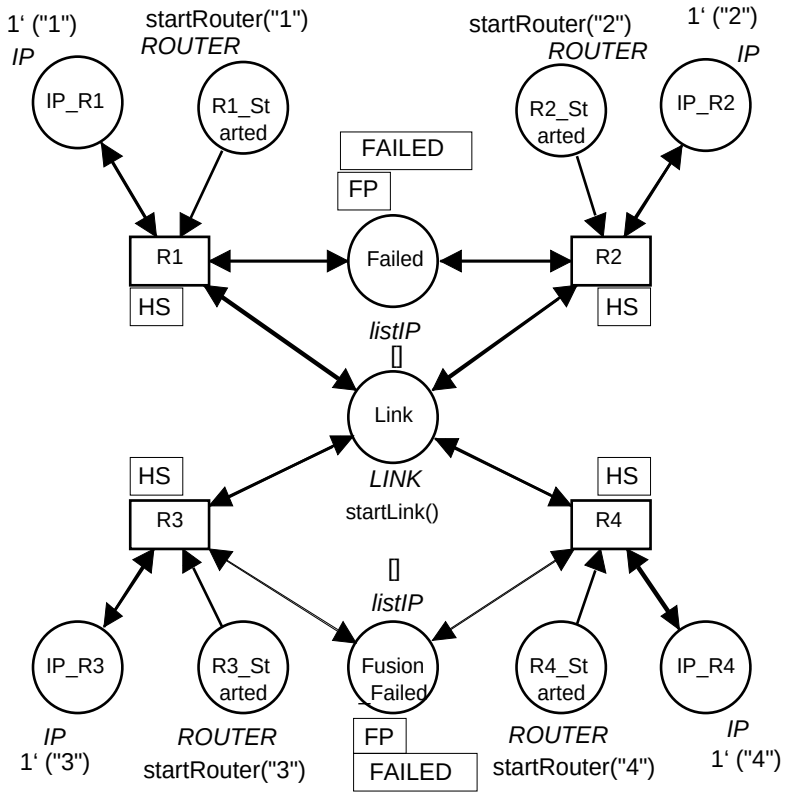


Figura C.2: Página *ConfInitialOSPF* – Página de mais alto nível

Na Figura C.2 é apresentada a página de Configuração Inicial do protocolo *OSPF*. Esta é a página de mais alto nível do modelo. As transições de substituição *R1, R2, R3* e *R4*

representam uma abstração de cada um dos roteadores da rede. Os lugares (*IP_R1, IP_R2, IP_R3* e *IP_R4*) armazenam fichas que representam os endereços IP de cada um dos roteadores. Os lugares (*R1_Started, R2_Started, R3_Started* e *R4_Started*) armazenam fichas que representam cada um dos roteadores em seu estado ativo. O lugar *Link* armazena fichas que representam cada um dos enlaces da rede. Cada enlace conecta dois roteadores, sendo que a rede modelada é completamente conectada, ou seja, existe um enlace conectando quaisquer dois roteadores da rede. Por fim, os lugares *Failed* e *Fusion_Failed* são o mesmo lugar representado neste modelo por dois lugares de fusão por uma questão de organização. Neste lugar existe uma ficha que mantém a lista dos roteadores que falharam e ainda não se recuperaram.

C.3 Página Router

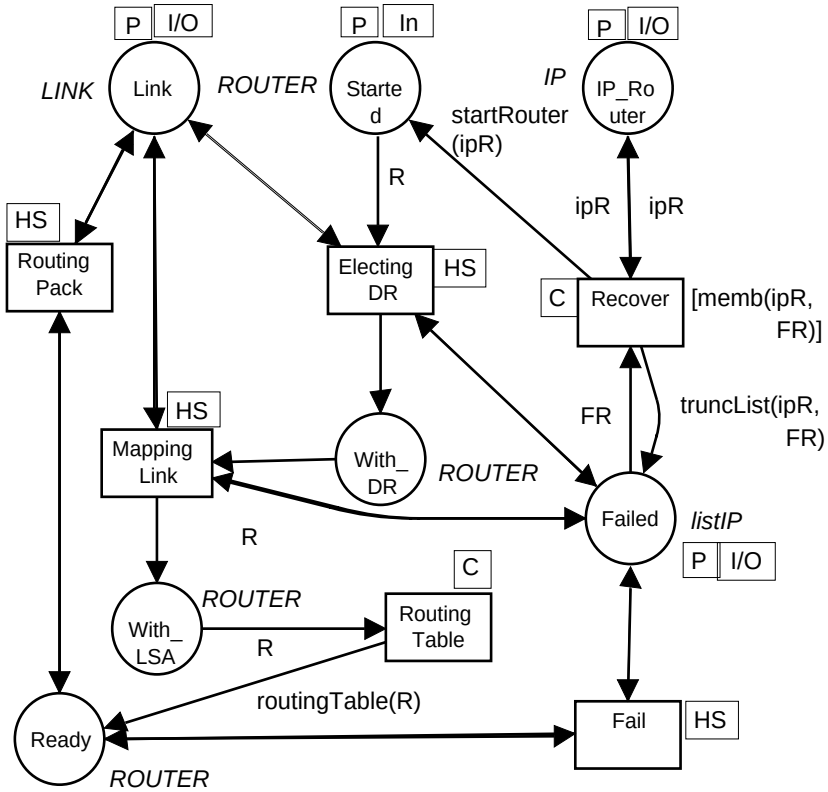


Figura C.3: Página *Router*

Na Figura C.3 é apresentada a página *Router*. Esta página representa o funcionamento geral do protocolo *OSPF* em cada roteador em um nível mais abstrato. Inicialmente, existe uma ficha no lugar *Started* indicando que o roteador está ativo e pronto para iniciar a exe-

cução do protocolo. Nesta etapa do protocolo o roteador deve eleger o roteador designado, que é o roteador responsável por receber os anúncios de estado de enlace (LSA's) dos demais roteadores e repassá-los para que todos os roteadores conheçam a topologia da rede. O processo envolvido na escolha do roteador designado é representado nesta página pela transição de substituição *Electing_DR*. Ao conhecer o roteador designado, os roteadores passam a trocar pacotes LSA's a fim de obter um mapa completo dos enlaces da rede. Este processo é representado pela transição de substituição *MappingLink*. Quando o roteador passa a conhecer o estado de todos os enlaces da rede, ele deve então calcular sua tabela de rotas. Isso é representado pela transição *Routing Table*. A função *routingTable* implementa o algoritmo de Dijkstra e é responsável por construir a tabela de roteamento do roteador. A partir do estado *Ready* o roteador está pronto para rotear pacotes de dados. O processo de roteamento de pacotes de dados é representado pela transição de substituição *Routing*. Neste ponto do funcionamento do protocolo o roteador pode também sofrer falhas, o que é representado pela transição de substituição *Fail*. Um roteador que tenha sofrido uma falha é inserido na lista armazenada no lugar *Failed* e pode se recuperar da falha, representado no modelo pela transição *Recover*. Ao se recuperar, o roteador volta ao estado ativo, o que é representado por uma ficha no lugar *Started*.

C.4 Página Router_DR

Na Figura C.4 é apresentada a página *Router_DR*. Esta página representa o processo de eleição do roteador designado. O roteador designado é o roteador responsável por receber e repassar os avisos de estado de enlace para os demais roteadores da rede. Para eleger o roteador designado, os roteadores inicialmente trocam pacotes HELLO a fim de conhecer seus vizinhos ativos. A troca de pacotes HELLO é representada nesta página pelas transições *Send Hello* e *Receive Hello*. A guarda associada à transição *Receive Hello* garante que apenas as mensagens destinadas a um determinado roteador serão recebidas por este roteador. Além disso, a guarda garante também que se existe alguma falha não percebida o roteador fica impedido de enviar ou receber pacotes. Após receber mensagens de todos os seus vizinhos, o roteador pode então escolher o roteador designado. A escolha do roteador designado é representada no modelo pela *transição Electing DR*, estando a função *electDR* responsável por realizar tal escolha. A eleição é baseada nas prioridades dos roteadores da

rede, sendo escolhido aquele roteador de maior prioridade. As prioridades dos roteadores devem ser configuradas pelo administrador do sistema. Durante o processo de eleição do roteador designado, o roteador pode ainda perceber que algum roteador da rede sofreu uma falha. No modelo, a percepção de uma falha é representada pela transição *Get Fail*. A função *getFail* é responsável por atualizar os parâmetros do roteador para indicar que algum de seus vizinhos não está mais ativo.

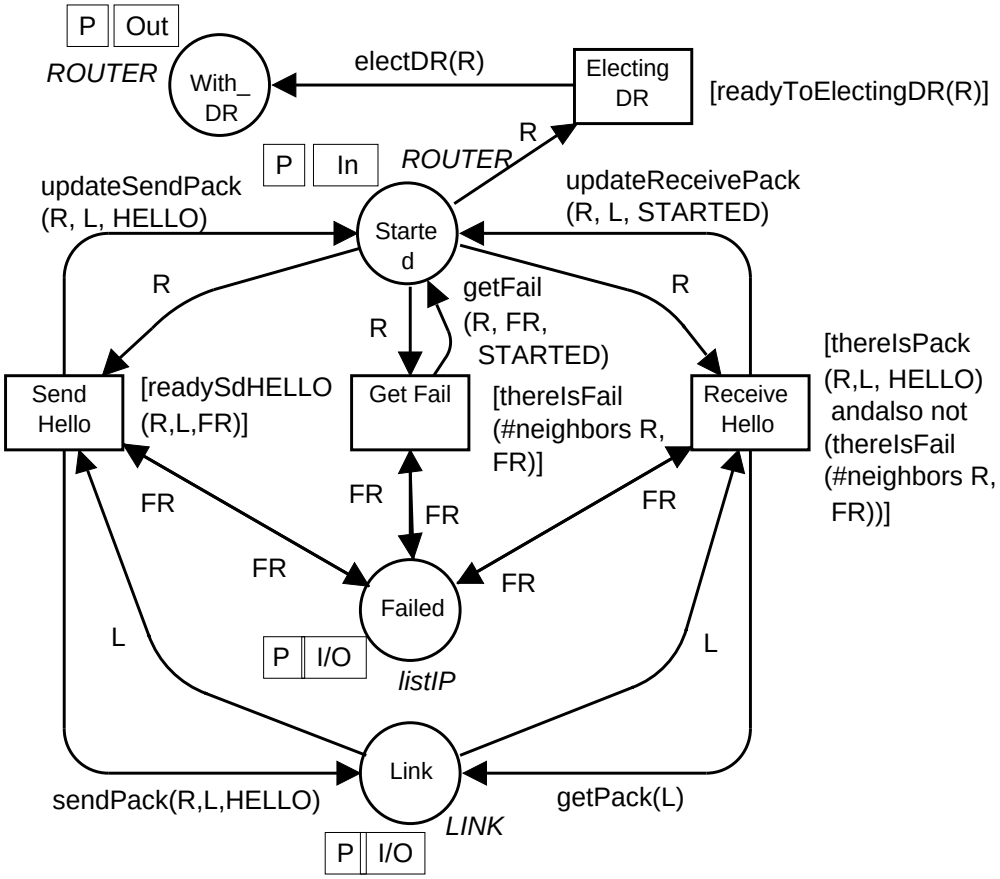


Figura C.4: Página *Router_DR* – Eleição do Roteador Designado

C.5 Página MappingLink

Na Figura C.5 é apresentada a página *MappingLink*. Esta página representa o processo de construção do mapa de enlaces da rede. A fim de montar este mapa, os roteadores trocam pacotes LSA's e atualizam suas bases de dados que contêm informações sobre a topologia da rede. O processo de atualização das bases de dados é representado no modelo pela transição de substituição *LSA*. Quando a base de dados está completa, o roteador constrói então um

mapa que descreve exatamente a topologia da rede, ou seja, quais são os enlaces ativos da rede e quais são os custos associados a estes enlaces. A construção deste mapa é representada no modelo pela transição *Mapping Link*. A guarda desta transição garante que a construção do mapa só acontece se a base de dados do roteador estiver completa, a fim de garantir que o mapa gerado esteja consistente com o estado atual da rede. Uma ficha no lugar *With LSA* indica que o roteador está com o mapa pronto. Durante o mapeamento dos enlaces o roteador pode perceber que algum outro roteador da rede sofreu uma falha. A percepção de falhas é representada no modelo pela transição *Get Fail*. O processo de percepção de falhas é idêntico em qualquer etapa do funcionamento do protocolo.

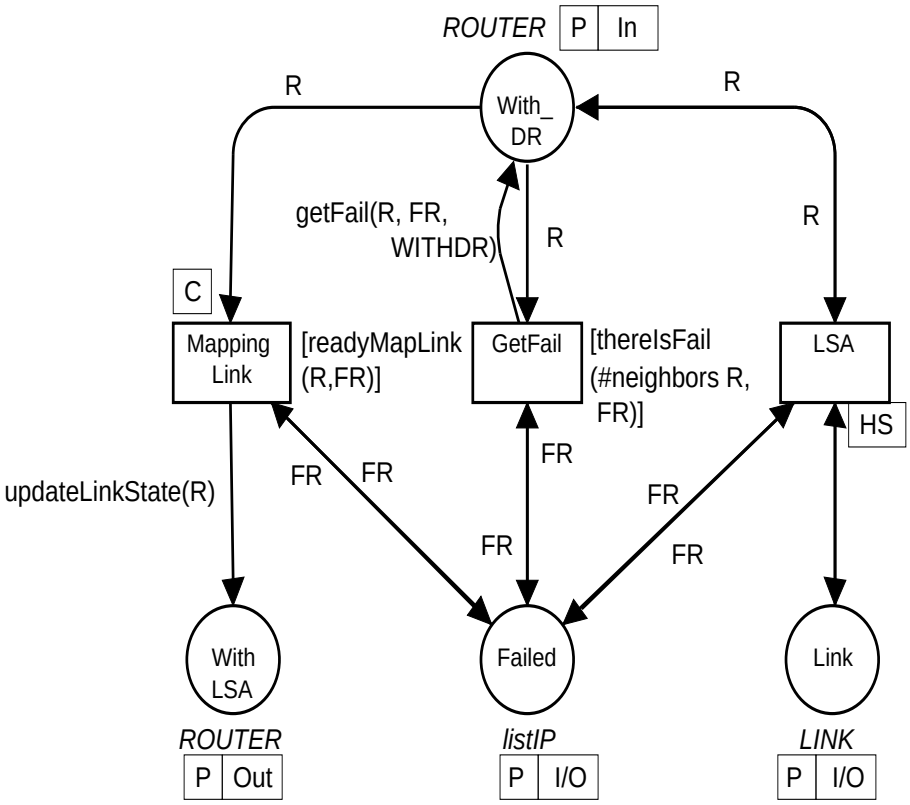


Figura C.5: Página *MappingLink* – Mapeando os enlaces para a troca de pacotes

C.6 Página SendReceiveLSA

Na Figura C.6 é apresentada a página *SendReceiveLSA*. O mapeamento dos enlaces da rede acontece depois que o roteador estiver com sua base de dados completa. A atualização desta base de dados se dá através da troca de pacotes LSA's.

Roteadores não designados devem enviar seus LSA's para o roteador designado. Já o roteador designado deve enviar seu LSA para os demais roteadores da rede. O envio dos pacotes LSA's é representado no modelo pela transição *Send LSA*. Além de enviar seu pacote LSA, o roteador designado deve também repassar os LSA's que ele recebeu dos outros roteadores. Essa atividade é representada pela transição *SendLSAallrts*. A última transição encontrada nesta página é a transição *Receive LSA* que representa o recebimento dos pacotes LSA's.

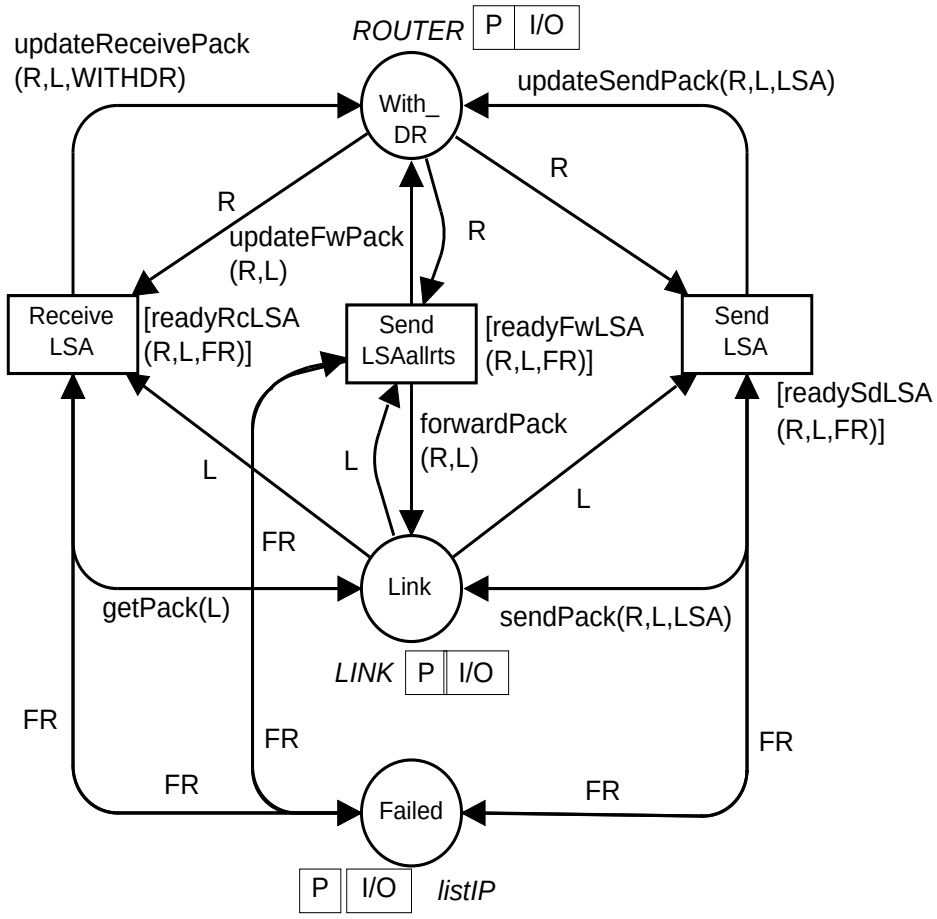


Figura C.6: Página *SendReceiveLSA* – Atualização da Base de Dados (Anúncio de Estado de Enlace – LSA's)

C.7 Página Routing

Na Figura C.7 é apresentada a página *Routing*. O processo de roteamento de pacotes de dados se dá da seguinte forma:

1. O roteador recebe um pacote. O recebimento de pacotes está representado nesta página pela transição *Receive Pack*.

2. O roteador analisa se ele é o destino final do pacote. Se ele é o destino final, então o pacote é removido do enlace, o que é feito pela função *getPack*, e consumido pelo roteador. Caso o roteador não seja o destino final do pacote, o pacote é inserido em uma lista de pacotes a enviar. Neste momento, a tabela de rotas é consultada para saber qual caminho o pacote deve seguir na rede. Todo este processo é executado pela função *updateReceivePack*.

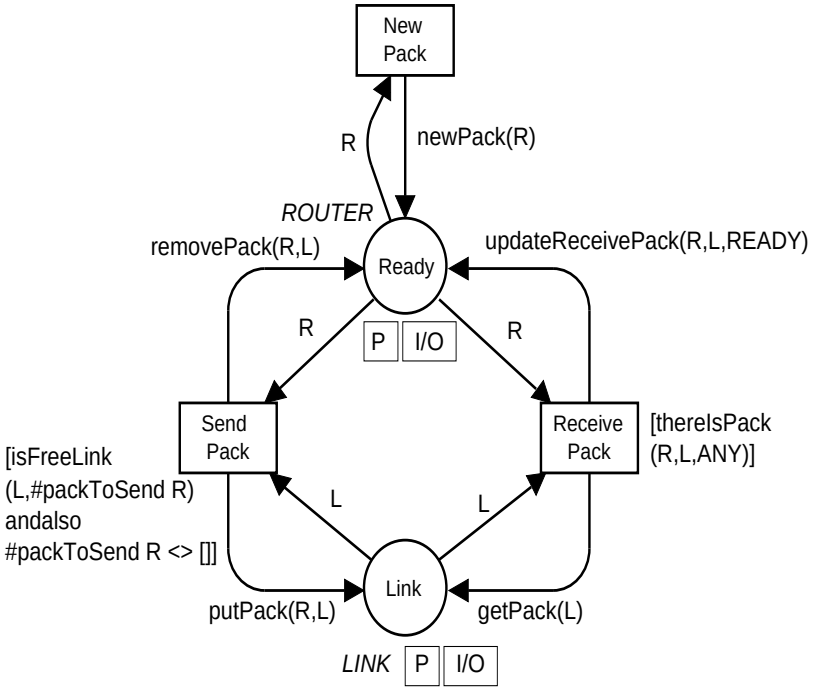


Figura C.7: Página *Routing* – Roteamento dos pacotes de dados

3. Por fim, os pacotes presentes na lista do roteador são enviados pelo enlace mais adequado. O envio dos pacotes é representado pela transição *Send Pack*. O enlace é atualizado para conter o pacote através da função *putPack*.

Os pacotes de dados neste modelo são gerados aleatoriamente. A geração dos pacotes é representada pela transição *New Pack* e é realizada pela função *newPack*. Novos pacotes são gerados apenas quando o roteador se encontra no estado *Ready* (pronto).

C.8 Página Failures

Na Figura C.8 é apresentada a página *Failures*. De posse da sua tabela de rotas, um roteador tanto pode rotear pacotes de dados quanto pode sofrer uma falha, representado no modelo pela transição *Fail*. Ao falhar, o roteador é inserido na lista de roteadores inativos armazenada no lugar *Failed*. Além de sofrer falhas e rotear pacotes, um roteador no estado pronto, representado por uma ficha no lugar *Ready*, pode também detectar falhas de outros roteadores. Este processo é representado pela transição *Get Fail*.

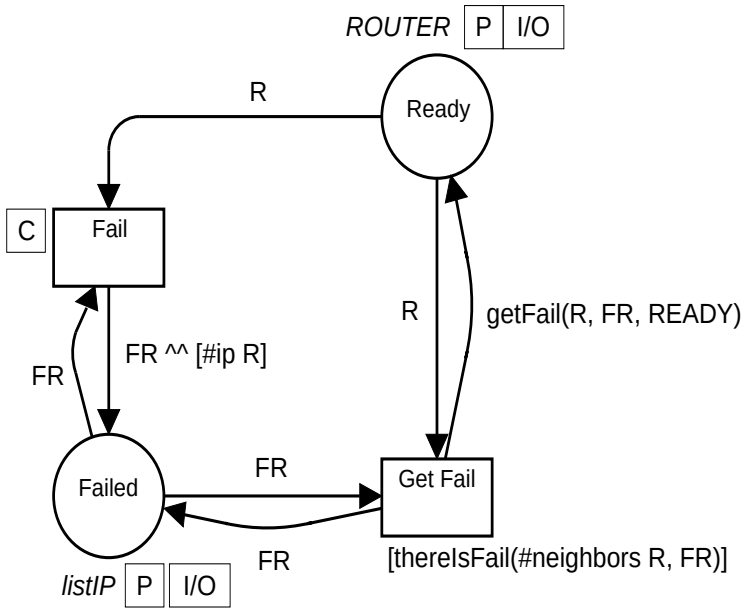


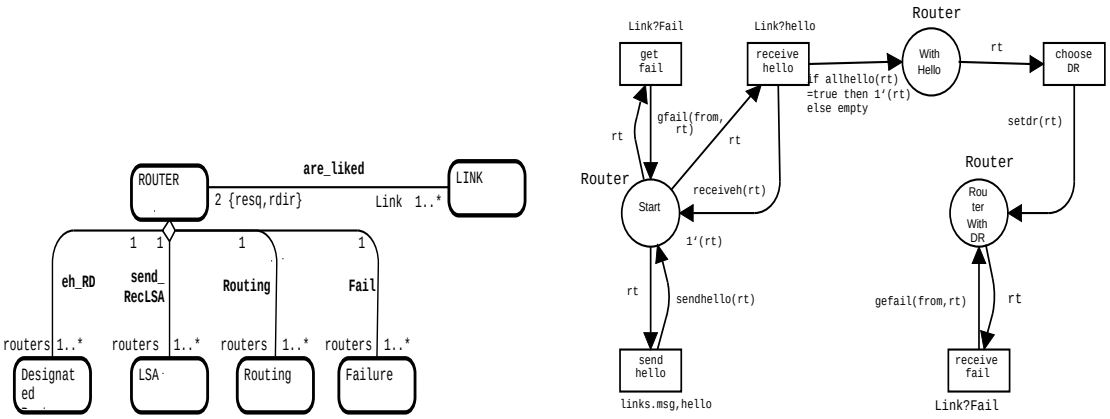
Figura C.8: Página *Failures* – Inserindo Falhas em um Roteador

C.9 Diagrama de Classes e Designated Router

C.10 Classe LSA

Na Figura C.10 é apresentado o corpo da classe *LSA*. No corpo desta classe temos representado o procedimento em que o router envia e recebe pacotes de anúncio de estado de enlace (pacotes *Description Database*). A transição *sendLSAtoDR* modela o envio da mensagem *GetLink(rt)!msg,DDatabase* para o roteador designado, o qual irá repassar para os demais roteadores ativos na rede.

A transição *sendLSAtorts* modela a situação na qual o roteador designado envia a mensagem *AllLink(rt)!msg,DDatabase* para os demais roteadores.



(a) Diagrama de Classes (b) Classe Designated Router

Figura C.9: *Diagrama de Classes* e *Classe Designated Router*

A transição *receive LSA* modela o recebimento dos pacotes *Description Database* através da mensagem *Link?DDatabase*. Quando os roteadores trocam estes pacotes, eles estão anunciando os seus estados de enlaces (vizinhos e custos). Com essas mensagens eles vão montando uma base de dados, composta por uma lista de todos roteadores e seus respectivos custos. Conhecendo todos os estados de enlace dos roteadores, situação na qual há uma ficha no lugar *Router With LSA*, o roteador poderá então calcular a sua tabela de rotas. O roteador calcula a tabela de roteamento aplicando o algoritmo de Dijkstra sobre a sua base de dados. Cada tabela é calculada localmente pelo roteador. Após o cálculo da tabela, teremos uma ficha no lugar *Router Ready*.

C.11 Classe Routing

Na Figura C.11 é apresentado o corpo da classe *Routing*. O corpo desta classe detalha o roteamento de pacotes de dados efetuado pelo router. A partir do estado *Router Ready* o roteador pode enviar ou receber um pacote de dados. A ocorrência da transição *receive_pack* modela o recebimento de um pacote de dados *Link?PacketData*. Caso o endereço IP de destino do pacote seja o do roteador, ele irá aceitá-lo e uma ficha será depositada no lugar *With Pack_data*, caso contrário, o pacote será enviado de acordo com a tabela de rotas (melhor caminho) ao seu destinatário – transição *send_pack*.

A ocorrência da transição *rec_LSA* indica que o roteador recebeu uma solicitação de

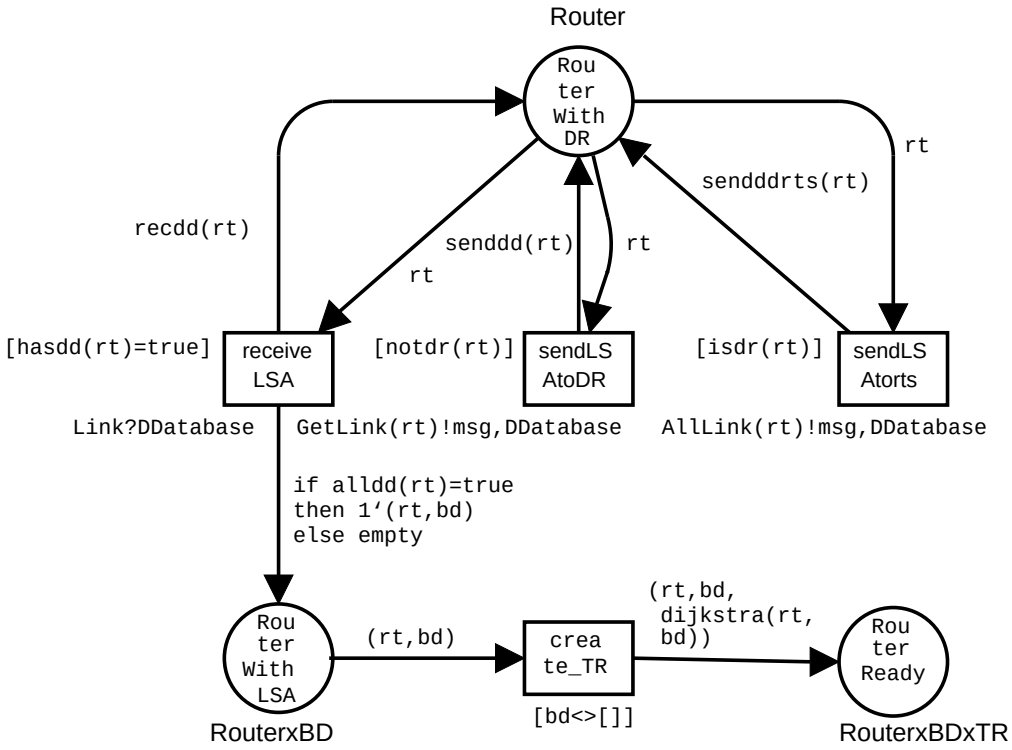


Figura C.10: Classe *LSA*

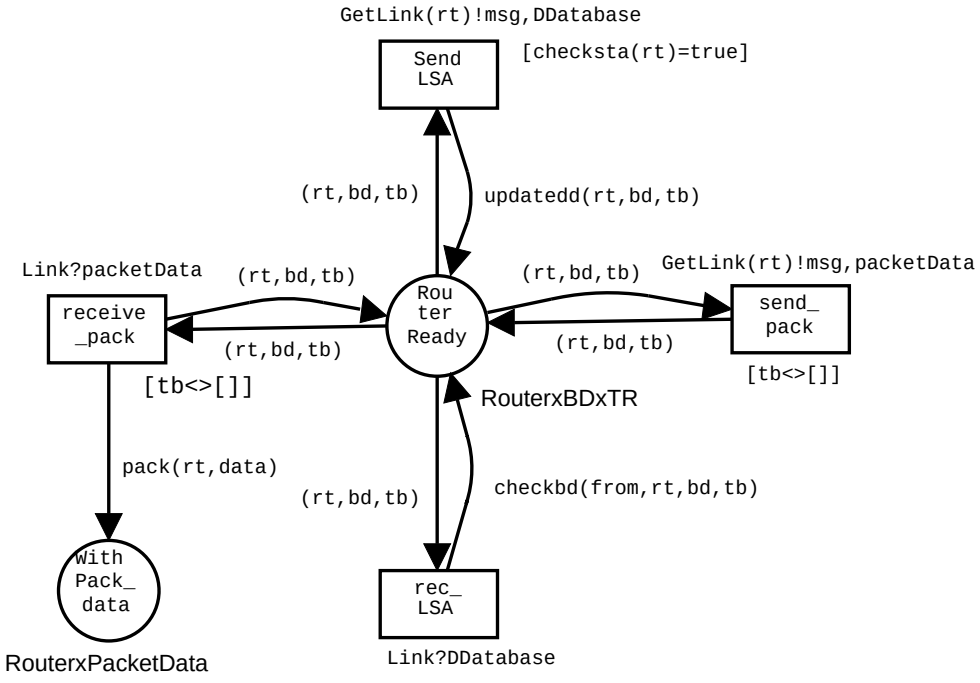


Figura C.11: Classe *Routing*

anúncio de estado de enlace de algum roteador que falhou e se recuperou (mensagem *Description Database*). Ao receber esta mensagem *Link?DDatabase*, o roteador verificará na sua base de dados a existência deste emissor, caso ele não esteja a mensagem *GetLink(rt)!msg,DDatabase* será enviada ao emissor (ocorrência da transição *send LSA*). O roteador continuará roteando pacotes de dados, mesmo quando estiver trocando pacotes *Description Database* para recalcular a sua tabela de rotas (Figura C.13) .

C.12 Classe Failures

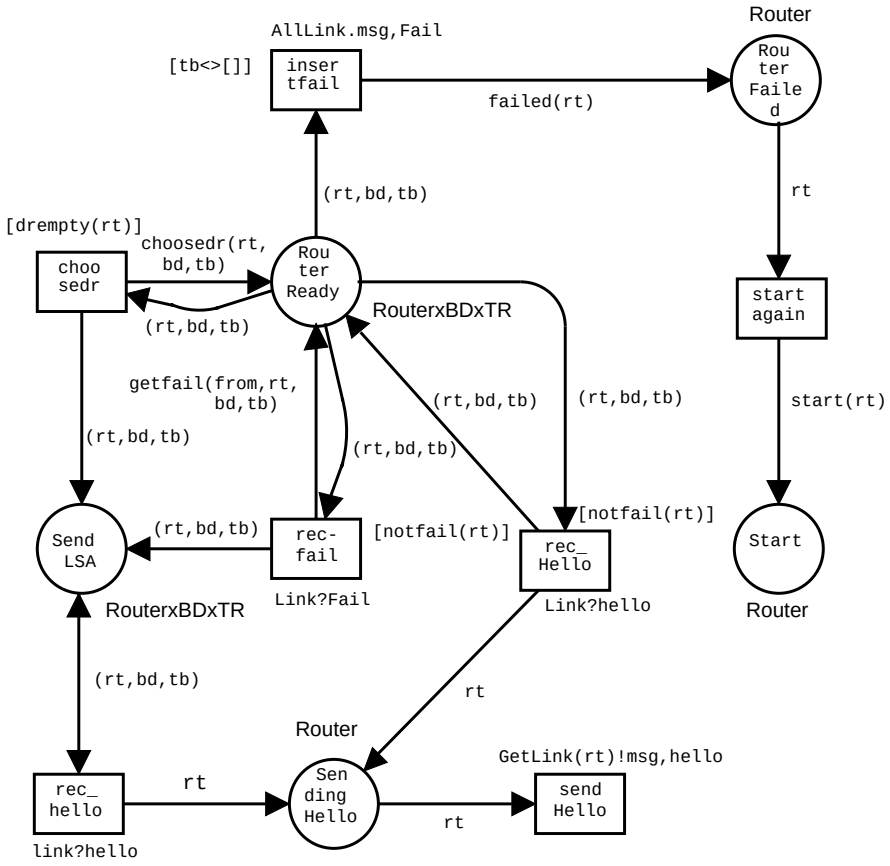


Figura C.12: Classe Failure

Nas Figuras C.12 e C.13 é apresentado o corpo da classe *Failure*. A partir do estado *Router Ready*, o roteador pode sofrer uma falha. O disparo da transição *insert fail*, modela a falha de um roteador. Quando um roteador falhar a mensagem *AllLink(rt)!msg,Fail* será enviada aos demais roteadores da rede avisando que o roteador falhou. Após, uma ficha será colocada no lugar *Router Failed*, onde a transição *start again* poderá ocorrer. A ocorrência

desta transição indica que o roteador será reinicializado, ou seja, voltará ao estado inicial, iniciando toda a troca de pacotes novamente.

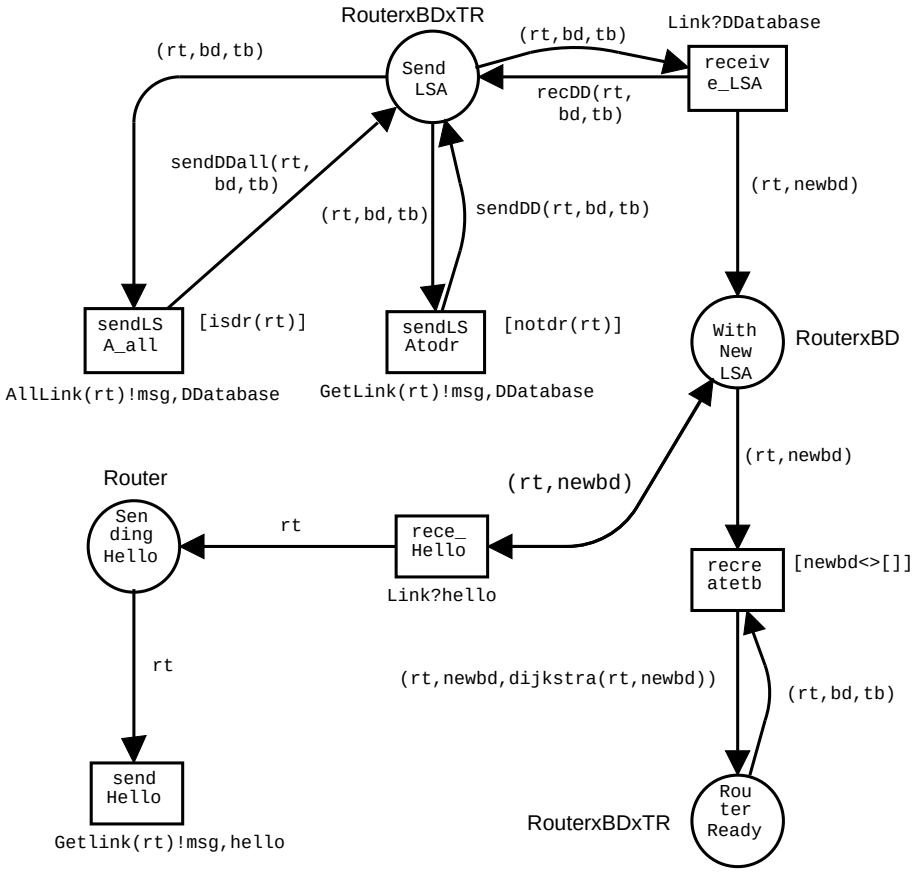


Figura C.13: Classe Failure

Ocorrendo a transição *rec_Hello*, a qual possui a inscrição de interação *Link?hello*, será depositado uma ficha no lugar *Sending Hello*, onde será enviado ao emissor uma mensagem *hello* como resposta. Isto é modelado pela inscrição de interação *GetLink(rt)!msg,hello* da transição *send Hello*.

O roteador pode receber uma mensagem *Link?Fail*. Ocorrendo a transição *rec_fail*, o roteador verificará se o roteador que falhou é o roteador designado, se for, a transição *choose dr* irá ocorrer para que seja efetuado a escolha do novo roteador designado. Após o recebimento da mensagem *Link?Fail* uma ficha será depositada no lugar *Send LSA*. Neste estado o roteador também pode receber a mensagem *Link?hello* novamente.

Quando o roteador detecta a falha de um roteador na rede (recebe a mensagem *Link?Fail*), ele irá enviar uma mensagem *Description Database* novamente, a fim de atualizar a sua

base de dados (Figura C.13). A partir do lugar *Send LSA*, podem ocorrer as transições *sendLSA_all*, *sendLSAtodr* e *reiceive_LSA*. A ocorrência destas transições indica que o roteador reiniciou o processo de enviar e receber mensagens *Description Database*, a fim de atualizar a sua base de dados após a falha de um dos roteadores da rede.

Após enviar e receber as mensagens *Description Database*, as quais contém os novos estados de enlace (nova base de dados) dos roteadores, uma ficha será depositada no lugar *With New LSA*. Neste estado podem ocorrer as transições *rece_Hello* e *recreate tb*. A ocorrência da primeira transição indica que o roteador recebeu a mensagem *Link?hello*. A ocorrência da segunda transição indica que o roteador possui uma nova base de dados, estando apto a recalcular a sua tabela de rotas de acordo com as alterações.

C.13 Classe LINK

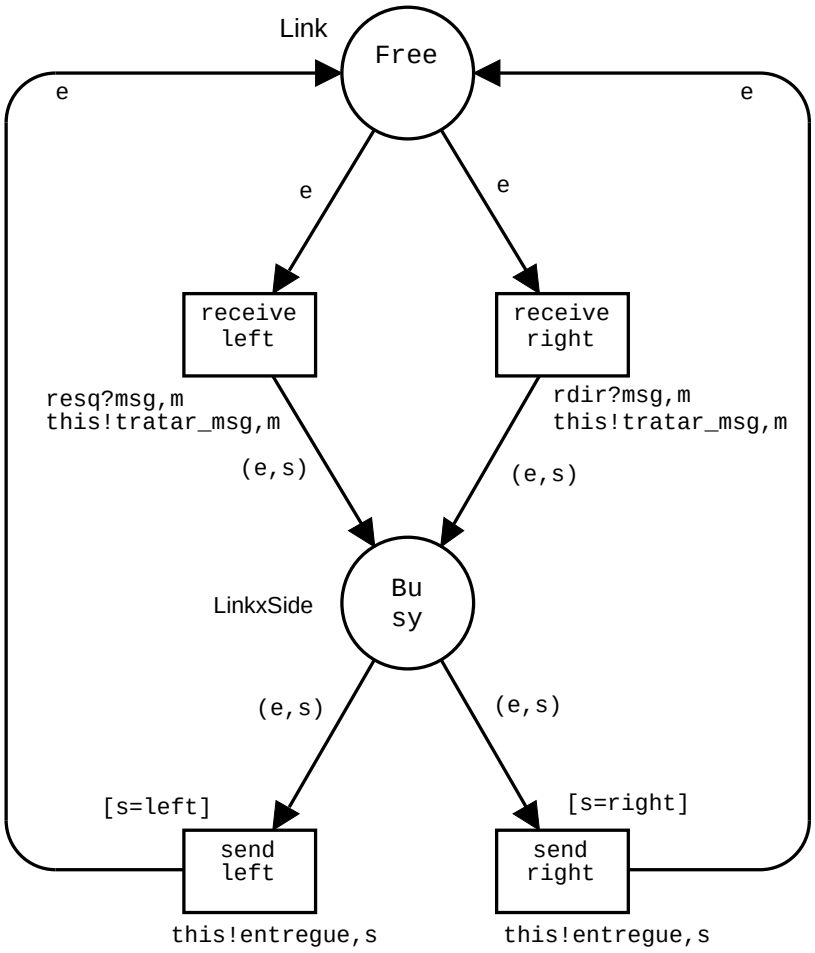


Figura C.14: Classe LINK

Nas Figuras C.14 e C.15 é modelado o comportamento da classe *LINK*. O objeto *link* conecta dois roteadores (roteadores da direita e esquerda), e tem dois estados: livre ou ocupado (lugares *Free* e *Busy*) da Figura C.14. Se o *link* estiver no estado *Free*, ele pode receber uma mensagem do roteador da esquerda (transição *receive left*) ou do roteador da direita (transição *receive right*. Ao receber uma mensagem de um roteador, o *link* irá fazer o tratamento desta mensagem (Figura C.15), e uma ficha será depositada no lugar *Busy*.

A partir deste estado, uma das seguintes transições podem ocorrer: *send left* e *send right*. A ocorrência das transições *send left* e *send right*, indicam que o *link* efetuou o tratamento da mensagem e está entregando-a ao roteador da esquerda/direita.

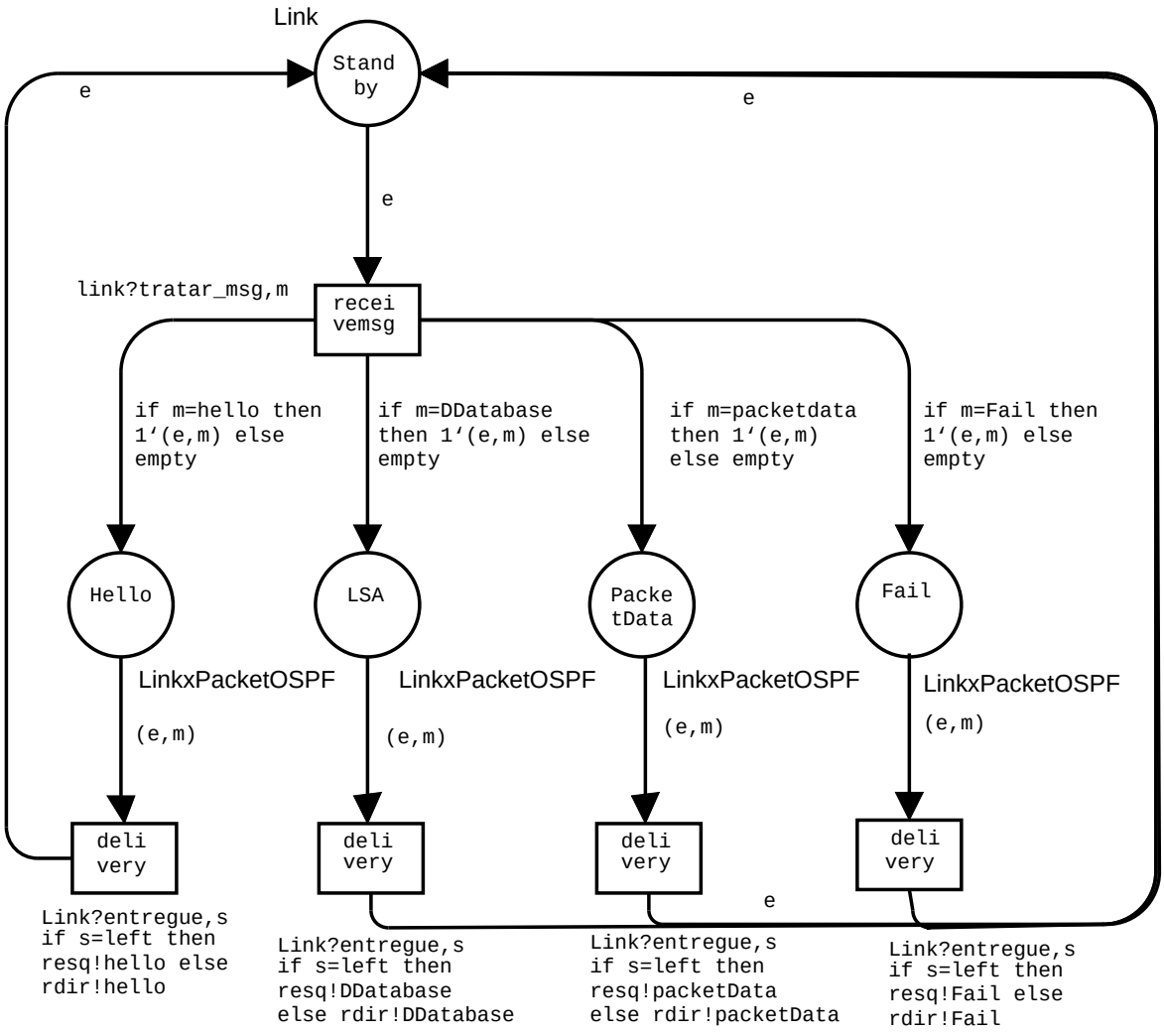


Figura C.15: Classe LINK - Tratamento das Mensagens

C.14 Diagrama de Configuração Inicial

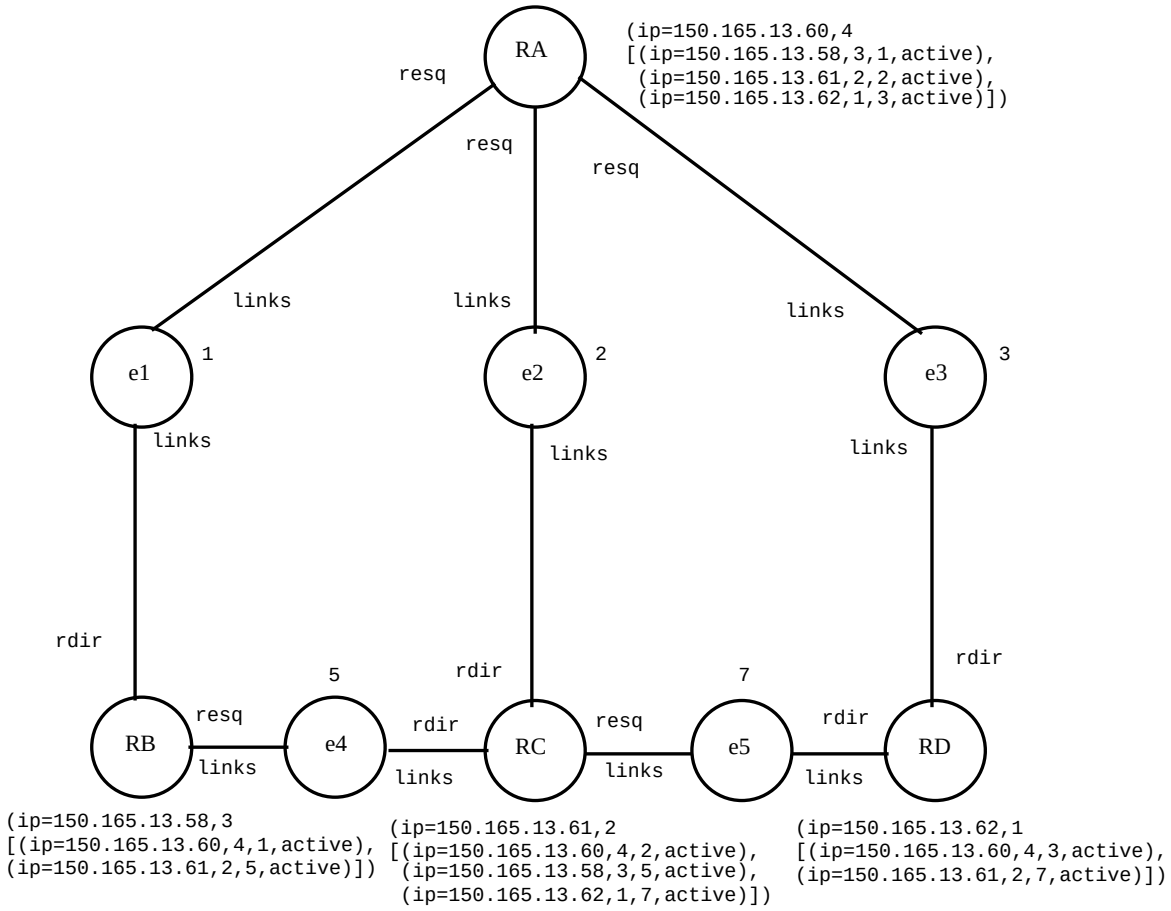


Figura C.16: Diagrama de Configuração Inicial