Igor Chaves Cananéa

Protomake: um compilador de protocolo

Orientador:
Djamel Hadj Fawzi Sadok

Universidade Federal de Pernambuco - Centro de Informática

Agradecimentos

À minha família, que sempre me incentivou para eu dar o melhor de mim; ao meu pai, maior incentivador, pelo seu jeito simples, humilde e prático, sempre me incentivando a ler e entender melhor o que eu estava fazendo; À minha mãe que, mesmo entendendo de informática tanto quanto eu entendo de doença de criança, sempre me escutou e aconselhou quando eu tinha alguma dúvida. Amo todos vocês.

Aos professores do Centro de Informática, em especial ao meu orientador Djamel Sadok e aos professores Judith Kelner, André Santos e Geber Ramalho, por terem sempre me incentivado a fazer o que eu gosto, transformando o trabalho em uma diversão.

Aos professores Paulo Andrade e Valdir Balbino e todos do Gentrop por terem me dado a oportunidade de trabalhar com eles e aprender a ver o mundo de um jeito diferente, sem bits nem bytes (mas ainda fazendo uso deles).

À minha noiva Karina Luna que aturou todas as minhas manias e estresses, sempre me fazendo rir e me sentir feliz com o seu jeito espontâneo, autêntico e sincero de ser. Amo você.

Aos meus colegas de faculdade, em especial aos meus amigos Mouse; Marcelo Jucá; Yzmof e Nacha; e do GPRT, em especial Josy e Manu, pelas conversas, discussões, farras e bebedeiras durante esses 5 anos de faculdade, além é claro do apoio que sempre tive de vocês quando mais precisei.

Ao Linus Torvalds pelo maravilhoso sistema operacional Linux, no qual todo esse trabalho foi desenvolvido, criado inicialmente por ele e desenvolvido hoje por uma comunidade enorme de hackers.

Aos criadores do mecanismo de busca Google. Sem ele, muitas pesquisas relevantes não poderiam ter sido realizadas a contento.

E por último, e não menos importante, à Deus, pai todo poderoso que nos deu a vida e o livre arbítrio, para escolher o que devemos fazer segundo nossas intenções com sensatez; ou não.

Resumo

O projeto e desenvolvimento de protocolos de comunicação são tarefas não triviais que demandam tempo e esforço. As ferramentas disponíveis atualmente não são suficientes para descrever protocolos complexos e, a partir da descrição, gerar aplicações que podem ser usadas para testar o protocolo na prática. Além disso, a depuração do protocolo é complicada, pois, mesmo utilizando o *Ethereal*, as mensagens de protocolos desconhecidos pela ferramenta (tendo em vista que, para os protocolos conhecidos, ela funciona adequadamente) são mostradas como seqüências de *bytes*, dificultando a visualização da informação. O objetivo deste projeto é apresentar um compilador de protocolos, Protomake, que, a partir da descrição do protocolo em uma linguagem de propósito específico, ETAP, gera aplicações usáveis na prática em uma linguagem alvo (C, Java, Python, etc) e um *plugin* de visualização para o *Ethereal*, facilitando a depuração.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Protocolos de comunicação são complexos e o grau de complexidade depende principalmente da sua aplicação. Implementar um protocolo pode ser considerado como uma tarefa não trivial, primeiramente porque o programador precisa ter um bom conhecimento da linguagem de programação que vai ser usada e, principalmente, necessita entender bem a especificação. Infelizmente, protocolos geralmente são especificados usando uma linguagem informal, deixando margem a interpretações nem sempre adequadas. Mesmo quando se usa as especificações formais, estas não possuem implementações livres de erros porque a tradução da especificação para um programa é, na maioria das vezes, realizadas por técnicos e os mesmos cometem erros. Ferramentas para auxiliar a codificação são bem-vindas, evitando erros comuns e dando um nível de abstração mais alto para o programador. Atualmente, as ferramentas disponíveis não são suficientes para descrever protocolos complexos e, a partir da descrição, gerar aplicações que podem ser usadas para testar o protocolo na prática.

Outra tarefa negligenciada pelas ferramentas é a depuração do protocolo. *Sniffers* de rede como o *Ethereal* [1] ajudam, mas possuem uma limitação séria: as mensagens de protocolos desconhecidos são descritas como seqüências opacas de *bytes*. Embora essa limitação seja contornável para protocolos simples, projetos mais complexos exigem uma forma de visualização mais inteligível das mensagens trocadas entre os pares. Dentro desse contexto, este trabalho apresenta uma ferramenta chamada Protomake

Protomake é um compilador de protocolos que facilita o desenvolvimento de novos protocolos e a criação de aplicações para protocolos já estabelecidos, aumentando o nível de abstração da especificação do protocolo, com a vantagem de utilizar uma linguagem simples e de propósito específico, ETAP [8], para especificar os protocolos. Essa ferramenta gera, a partir de uma descrição do protocolo em ETAP, um conjunto de funções e uma aplicação em uma linguagem alvo (no caso deste trabalho as linguagens alvo são C e Java). Para a linguagem C o suporte já foi concluído e está em desenvolvimento um suporte para a linguagem Java. O código gerado pode ser usado por outras aplicações em desenvolvimento e a aplicação gerada por Protomake é um exemplo que pode ser usado para avaliar se a comunicação está funcionando adequadamente através do protocolo descrito.

Por ser simples e sucinta, ETAP permite ao programador preocupar-se apenas com o funcionamento do protocolo e não como essas funcionalidades serão implementadas. Para facilitar a implementação de funcionalidades que não fazem parte do protocolo diretamente, existe um mecanismo para utilizar funções criadas na linguagem alvo dentro do protocolo (descrito em 4.2). O compilador se encarrega de gerar o código necessário para fazer a integração entre a descrição compilada e o código na linguagem alvo. Para facilitar a depuração do protocolo, Protomake pode gerar um *plugin* de visualização para o analisador de protocolos *Ethereal*, transformando a seqüência de *bytes* em uma descrição mais elaborada das mensagens.

Este trabalho está organizado da seguinte forma: no Capítulo 2 são descritos os trabalhos relacionados e que de alguma forma contribuíram para a criação do Protomake ou que possuem propósito semelhante; no Capítulo 3 a linguagem ETAP é descrita de forma detalhada; o Capítulo 4 apresenta o compilador criado; finalmente, no Capítulo 5 os comentários finais e as propostas de trabalhos futuros para o Protomake são descritos.

2 Trabalhos Relacionados

Várias linguagens e ferramentas foram desenvolvidas para facilitar o processo de desenvolver um protocolo de comunicação. Linguagens com SDL [7], LOTOS [6] e Promela [9] são usadas para especificar formalmente um protocolo. A principal vantagem delas é permitir a verificação e validação do protocolo especificado, dando garantias ao desenvolvedor de que o protocolo funciona como previsto. No entanto, converter essa especificação em código não é uma tarefa simples, porque elas possuem construções demasiadas abstratas para serem convertidas em um programa implementado em uma linguagem de programação.

Um caso especial é Promela++ [4], que pode ser considerado como um meio termo entre linguagens de especificação formal e linguagens de propósito específico usada para facilitar a programação de um protocolo. Embora exista uma descrição de um compilador para a linguagem, não foi possível obter uma cópia dele para que fosse possível avaliar a usabilidade da linguagem.

Linguagens como Esterel [5], Prolac [10], Morpheus [2] e TAP [12] têm como foco a geração de código a partir de uma descrição do protocolo para facilitar o desenvolvimento de uma aplicação que utilize um protocolo de comunicação. Existe também uma alternativa que usa XML e XLST [3] com esse mesmo fim. Nas seções seguintes, as linguagens Esterel, Prolac e TAP são descritas em maior detalhe.

2.1 Esterel

Esterel é uma linguagem para programar sistemas reativos e síncronos, também utilizada para validação de protocolos. Permite realizar a verificação e validação de sistemas reativos e o seu compilador gera sistemas para simular o protocolo criado. Esterel pode ser utilizada para gerar implementações de programas reativos em *software* ou *hardware*. O compilador de Esterel também gera código C para ser incorporado como núcleo de um programa maior.

Embora possa ser usada para construir protocolos de comunicação para redes de computadores, Esterel não é uma linguagem projetada especificamente para isso, pois não é possível especificar as mensagens diretamente na linguagem. Além disso, todo o código para realizar marshalling, unmarshalling, envio e recepção das mensagens deve ser escrito na linguagem C para posterior integração com o código gerado pelo compilador de Esterel.

2.2 Prolac

Prolac é uma linguagem orientada a objetos criada para desenvolver protocolos de rede. O objetivo da linguagem é facilitar o desenvolvimento de protocolos sem um compromisso com a corretude do mesmo, deixando isso a cargo do desenvolvedor. Embora seja possível implementar protocolos complexos com Prolac [11], a linguagem não possui construções para o envio e recepção de mensagens, que devem ser implementados na linguagem alvo, obrigatoriamente C. Também não há suporte para manipulação de temporizadores e, assim como em Esterel, todo o código de marshalling e unmarshalling deve ser implementado em C.

2.3 TAP

TAP (*Timed Abstract Protocol*) é a linguagem de especificação de protocolos do APC (*Austin Protocol Compiler*) [12]. Foi desenvolvida para facilitar a criação de protocolos da camada de aplicação que utilizam UDP como camada de transporte. Tanto as mensagens como o protocolo podem ser descritos usando a linguagem e o compilador, o APC, gera todo o código necessário para criar uma aplicação que use o protocolo especificado. O código gerado é em C e é possível usar funções escritas em C diretamente na especificação do protocolo. A sintaxe da linguagem é simples, baseada em guardas que, caso sejam satisfeitas, executam uma seqüência de ações.

Embora TAP seja a única linguagem entre as descritas neste trabalho que pode gerar uma aplicação completa apenas a partir da especificação do protocolo, ela possui algumas limitações como a geração de código apenas na linguagem C e a impossibilidade de enviar mensagens utilizando TCP como o protocolo da camada de transporte.

3 A linguagem ETAP

Para especificar um protocolo que possa ser transformado em código pelo Protomake, deve-se usar a linguagem ETAP (*Extended* TAP), que foi criada como uma extensão de TAP para suprir as suas deficiências. ETAP é uma linguagem de propósito específico e, portanto, mais simples do que as linguagens de propósito geral. Isso permite ao programador concentrar-se apenas na definição do protocolo e não em como ele vai ser implementado na linguagem alvo.

ETAP é uma linguagem simples para especificar protocolos da camada de aplicação através de guardas. Uma guarda é uma construção que possui uma condição (expressão booleana, recepção de mensagens ou *timeout*) e um conjunto de ações que serão executadas caso a condição seja satisfeita. Estas guardas são agrupadas em uma entidade chamada processo. As mensagens utilizadas pelo protocolo são descritas por uma estrutura de campos que podem ter os tipos *bits* ou *bytes* e podem ser enviadas tanto via TCP ou UDP. Um protocolo em ETAP é então um conjunto de mensagens e processos. Para especificar um protocolo, 4 (quatro) seções, das quais apenas 2 (duas) são obrigatórias, devem ser definidas, necessariamente nessa ordem: [IMPORTS], MENSAGENS, [BEHAVIORS] e PROCESSOS.

3.1 Operadores, símbolos e construções de ETAP

Os operadores e símbolos que podem ser utilizadas em ETAP são apresentadas na tabela 1, assim como os construções na tabela 2. Esses serão mais detalhadas no decorrer deste trabalho.

Operador / Símbolo	Descrição
+, -, *, /	Operadores aritméticos de adição (+), subtração (-), multiplicação (*) e divisão (/).
&, |, ~	Operadores lógicos de E (&), ou (|) e NÃO (~).
>, <, >=, <=, =, <>	Operadores de comparação menor (<), maior (>), menor ou igual (<=), maior ou igual (>=), igual (=) e diferente (<>).
( )	Parênteses, utilizados para agrupar sub-expressões aritméticas ou lógicas.
:=	Operador de atribuição.
[]	Operador condicional. Indica alternativa, caso uma condição booleana ou guarda não seja satisfeita.

Tabela 1: Operadores usados em protomake.

3.2 Imports

Imports são diretivas que importam o código ETAP que está em outros arquivos para que possam ser utilizados pelo arquivo corrente e devem ser declarados como no exemplo da figura 1.

```
import "iccp-messages.pm"
```

Figura 1: Exemplo de importação de arquivo.

A maior vantagem de importar arquivos é a modularização da especificação do protocolo, pois partes do protocolo podem ser definidos em arquivos diferentes, facilitando a manutenção e a atualização do mesmo.

3.3 Mensagens

Mensagens são definidas como registros onde os tipos de cada campo podem ser *bits* ou *bytes*. O tipo *bits* é usado para representar valores inteiros de até 32 (trinta e dois) *bits* com sinal. Já o tipo *bytes* é usado para representar seqüências de *bytes* ou *strings* (seqüências de caracteres) e ao menos uma mensagem deve ser especificada.

```
message msg
begin
    nome_campo : n bits | bytes [ = init ]
    ...
end
```

Figura 2: Código para especificar mensagens.

Como descrito na figura 2, uma mensagem começa com a palavra chave **message** , seguida pelo nome dado a mensagem (nesse caso `msg`). Em seguida, a palavra chave **begin** e os campos das mensagens, onde ao menos 1 (um) campo deve ser definido. Por último, a palavra chave **end**, indicando o fim da mensagem. Uma mensagem pode ter quantos campos forem necessários. Um campo pode conter uma inicialização opcional (`init`) que pode ser um inteiro (*bits*) ou um *string* (*bytes*). Toda vez que a mensagem for enviada, os campos que foram inicializados devem conter o valor especificado. Caso contrário, a mensagem é descartada na recepção.

Para cada campo, a quantidade (`n`) de *bits* ou *bytes* deve ser especificada. Para *bits*, `n` deve ser um valor entre 1 e 32. Para *bytes*, `n` pode ser um valor inteiro entre 1 e $2^{32} - 1$ ou um campo do tipo *bits* previamente definido. A figura 3 mostra esse caso, onde a quantidade de *bytes* será o valor que estiver armazenado em `nome_campo1` no momento em que a mensagem for enviada e este valor será um inteiro entre 1 e $2^{10} - 1$.

```
message msg
begin
    nome_campo: 10 bits
    nome_campo1: nome_campo bytes
end
```

Figura 3: Campo com tamanho definido por outro campo.

Não é possível especificar qual a ordem de chegada das mensagens do protocolo, tornando a recepção de mensagens não-determinística. Para poder reconhecer qual mensagem foi enviada, o compilador acrescenta à cada uma um campo de identificação. Esse campo é checado quando a mensagem chega e é utilizado para iniciar a recepção corretamente. As implicações desse campo na criação de aplicativos que usam protocolos conhecidos são descritas em 4.1.1.

3.4 Behaviors

Behaviors são construções que representam código externo, implementado na linguagem alvo, e que podem ser usados como ações em um processo. A definição de *behaviors* não é obrigatória, caso não haja código externo referenciado em algum processo. Um *behavior* define um módulo externo e deve ser especificado como ilustrado na figura 4.

Para cada *behavior* definido, é possível usar construções do tipo *behavior.função(parâmetros)* como uma ação válida. O compilador, ao encontrar essa construção, gera o código

```
behavior behavior1, ...
....
```

Figura 4: Declaração de behaviors.

necessário para que a ligação com o código externo seja feita corretamente. Cabe ao programador escrever o código de cada *behavior.função(parâmetros)* para que ele seja usável dentro do protocolo. *Behaviors* não retornam valor e podem receber qualquer número de parâmetros (inclusive nenhum), que são sempre passados por referência. As funções dentro de um mesmo behavior devem possuir nomes diferentes e são sensíveis a caixa, ou seja letra maiúscula e minúscula são diferentes (*behavior.foo* é diferente de *behavior.Foo*).

3.5 Processos

Processos são descrições das seqüências de ações desempenhadas pelo protocolo. Cada protocolo deve possuir ao menos 1 (um) processo, chamado *main*.

```
process name
[var | const a1, ..., aX : tipo [ = init ]]
...
begin
    guarda -> ações
    [ [] guarda -> ações ]
end
```

Figura 5: Declaração de um processo.

Como descrito na figura 5, processos começam com a palavra chave **process** seguida de um nome (nesse caso identificado por `name`). Em seguida, variáveis e constantes podem ser declaradas localmente ao processo, caso sejam necessárias, com as suas respectivas palavras chaves (**var** para variáveis, **const** para constantes), uma seqüência de nomes, um tipo e uma inicialização opcional (`init`). Os tipos válidos são descritos na tabela 3.

Caso não sejam inicializadas, as variáveis do tipo **integer**, **range N..M**, **string** e **array** são inicializadas com *0*, *N*, *string vazio* ("") e cada elemento como a inicialização padrão do seu tipo, respectivamente. Constantes devem ser obrigatoriamente inicializadas.

Atualmente, ETAP possui suporte para multi-processos, através das palavras chaves **start** e **stop**. Com isso, é possível iniciar processos que executaram em paralelo com o processo *main*, facilitando a criação do aplicações que usavam uma versão multi-processo de um protocolo. O compilador, no seu estágio atual de desenvolvimento, ainda não gera todo o código necessário para que multi-processos possam ser usados completamente.

3.5.1 Address, client_address e server_address

Os tipos **address**, **client_address** e **server_address** (referidos em conjunto como os tipos *address*) são especiais, pois eles representam os canais de comunicação usados pelo protocolo. A forma como eles são declarados (variáveis ou constantes) altera as suas semânticas.

Quando declarados como constantes, os tipos *address* identificam um endereço específico que é checado em cada recepção. Se o endereço de quem enviou a mensagem não for o esperado, a mensagem é descartada. A figura 6 ilustra um exemplo.

```
# Exemplo de um cliente de echo

message teste
begin
    ...
end

behavior console_IO

process main
const endereco : address
var start : integer = 1
begin
    start = 1 -> start := 0
                 send teste to endereco

    [] rcv teste from endereco -> console_IO.print(teste)
                                  stop
end
```

Figura 6: Cliente de echo.

Quando a mensagem `teste` é recebida, o endereço de origem é checado. Somente se for igual ao especificado para `endereco`, a ação correspondente será executada. Quando constantes, **address** e **server_address** são usados por clientes UDP e TCP, respectivamente, pois clientes normalmente conectam-se a servidores específicos. Já quando **client_address** é constante, é usado por servidores TCP para verificar se um cliente específico está enviando mensagens. O modo de definir os endereços constantes é descrito em 4.4

Apenas os tipos **client_address** e **address** podem ser declarados como variáveis.

Quando o são, armazenam o endereço de quem enviou uma determinada mensagem. Esse endereço pode ser usado para, posteriormente, enviar mensagens para o endereço, como mostra o exemplo da figura 7.

```
# Exemplo de um servidor de echo

message teste
begin
    ...
end

process main
var endereco : client_address
begin
    rcv teste from endereco -> send teste to endereco
end
```

Figura 7: Servidor de echo.

Quando a mensagem `teste` é recebida, a ação correspondente é imediatamente executada. No exemplo da figura 7, a mensagem é simplesmente repassada de volta para quem a enviou. **address** e **client_address** variáveis são usadas por servidores UDP e TCP, respectivamente. **server_address** variáveis não podem ser declarados porque, semanticamente, não faz sentido um cliente enviar mensagens para um servidor que ele não conhece *a priori*.

3.5.2 Guardas

Como foi dito anteriormente, um protocolo é definido em ETAP por suas mensagens e seus processos. Cada processo contém um conjunto de guardas, que são compostas de uma condição e um conjunto de ações que são executadas, caso a condição seja satisfeita. Os tipos de condições válidas são expressões booleanas, recepção de mensagens e *timeouts*.

As guardas com expressões booleanas são ativadas se a expressão for satisfeita. Essas tem precedência em relação as guardas com recepção de mensagens. As expressões booleanas são montadas usando as variáveis declaradas localmente ao processo e os operadores = (igual), <>(diferente), > (maior), < (menor), <= (maior ou igual), <= (menor ou igual), & (E lógico), | (OU lógico) e ~ (NÃO lógico). Parênteses podem ser usados para agrupar sub-expressões.

As guardas com recepção de mensagens são ativadas quando a mensagem especificada for recebida. Se o endereço for variável, apenas a mensagem é checada, caso contrário, o endereço de origem também é checado.

As guardas com *timeout* são ativadas quando esgota o seu tempo de espera. Os *timeouts* são iniciados quando uma ação **act** é executada e tem precedência sobre as outras guardas. Se mais de uma ação **act** for ativada para um mesmo *timeout*, o último act executado é considerado. A figura 8 ilustra esse caso.

```
# Exemplo usando os 3 tipos de guardas

message teste
begin
    ...
end

process main
var endereco : client_address
var a : integer
begin
    rcv teste from endereco -> send teste to endereco
                               a := 5
                               act A in 5000

    [] a > 0 & a < 10 -: a := 10
                         act A in 10000

    [] timeout A -> stop
end
```

Figura 8: Exemplo usando timeouts.

Timeouts podem ser cancelados com a operação **cancel** *timeoutName*, onde *timeoutName* é o nome do timeout que será cancelado.

3.5.3 Ações válidas

ETAP possui um conjunto mínimo de ações válidas que são necessárias para especificar os protocolos. Ações adicionais são especificadas através de behaviors, que devem ser implementadas pelo programador na linguagem alvo. A tabela 4 descreve as ações que são consideradas válidas em ETAP.

Campos de mensagens podem ser acessados através do operador ponto. Por exemplo, se uma mensagem `teste` foi definida com o campo `c1`, referências a esse campo podem ser feitas na forma`teste.c1`. Para exemplos de código escritos em ETAP, consulte o apêndice *A*.

Construção	Descrição
act timeout **in** time	Ativa o temporizador *timeout* para ser executado em *time* milisegundos.
address	Canal de comunicação que usa UDP como protocolo de transporte
array [size] **of** Type	Uma seqüência contígua de variáveis ou constantes do tipo *Type*.
begin ... **end**	Delimitadores de blocos. Usado por processos e mensagens.
behavior	Especifica um módulo externo contendo funções que podem ser usadas pelo protocolo.
bits	Tipo de dado usado para especificar valores inteiros em uma mensagem.
bytes	Tipo de dado usado para especificar uma seqüência de *bytes* ou um *string* em uma mensagem.
cancel timeout	Cancela um *timeout* que foi previamente ativado pela construção **act**
const	Declara constantes que serão usadas dentro de um processo.
client_address	Canal de comunicação que usa TCP como protocolo de transporte
do ... **od**	Delimitadores de laço de repetição.
if ... **fi**	Delimitadores de condicionais.
import	Importa código escrito em ETAP que está em outro arquivo.
integer	Tipo de dado usado para especificar variáveis ou constantes que armazenam valores inteiros em um processo.
message	Declara mensagens reconhecidas por um protocolo.
process	Cria um processo, que representa o funcionamento do protocolo
range N .. M	Tipo de dado usado para especificar variáveis ou constantes que armazenam valores inteiros entre N e M em um processo.
rcv msg **from** peer	Recebe um mensagem (nesse caso, msg) de um peer.
send msg **to** peer	Envia uma mensagem (nesse caso, msg) para um peer.
skip	Construção que instrui o protocolo a fazer nada.
server_address	Canal de comunicação que usa TCP como protocolo de transporte.
start	Inicia um novo processo.
stop	Finaliza o processo corrente.
string	Tipo de dado usado para especificar variáveis ou constantes que armazenam *strings* em um processo
timeout T	Um temporizador T que pode ser ativado através de uma construção **act**.
var	Declara variáveis que serão usadas dentro de um processo.

Tabela 2: Construções válidas em ETAP.

Tipo	Descrição
integer	Um valor inteiro de 32 bits.
range N..M	Um inteiro maior ou igual a N e menor ou igual a M.
string	Uma seqüência de caracteres
array [X] of **Type**	Um array de tamanho X do tipo *Type*. Type pode ser um **integer**, **range**, **string** ou outro **array**.
address	Uma abstração para endereços UDP. Não pode ser inicializado.
client_address	Uma abstração para endereços de clientes TCP. Não pode ser inicializado.
server_address	Uma abstração para endereços de servidores TCP. Não pode ser inicializado.

Tabela 3: Tabela de tipos válidos em ETAP.

Ação	Descrição
Atribuição	Realizada através do operador :=, a atribuição pode ser simples ou múltipla.
Envio de mensagem	A sintaxe para envio é: `send message to addr`, onde `message` é uma mensagem definida para o protocolo e `addr` é um endereço válido, declarado dentro do escopo do processo.
Condição	A sintaxe para condição é: `if boolean -> Acoes [ [] boolean -> Acoes ] ... fi`
Laço de repetição	A sintaxe para laços é: `do boolean -> Acoes od`
Faz nada	A palavra chave **skip** faz nada. É útil quando, dado uma guarda, se a condição for satisfeita, nenhuma ação deve ser tomada.
Ativar timeout	A sintaxe é: `act timeoutName in time`, onde `timeoutName` é o nome de uma guarda com condição de *timeout* e `time` é um tempo em milisegundos.
Cancelar timeout	A sintaxe é: `cancel timeoutName`, onde `timeoutName` é o nome de uma guarda com condição de *timeout* que foi previamente ativada.
Iniciar um novo processo	A sintaxe é: `start processName` onde `processName` é o nome de um processo válido.
Terminar o processo corrente	Realizada através da palavra chave **stop**. Se o processo corrente for o *main*, o motor do protocolo é finalizado.

Tabela 4: Ações válidas em ETAP.

4 O compilador Protomake

Atualmente, o compilador possui suporte de bom nível para a geração de código na linguagem C e está em desenvolvimento o suporte para a linguagem Java. O Protomake foi desenvolvido de modo que desenvolver o suporte para uma determinada linguagem alvo fosse fácil. A descrição do código gerado criado pelo Protomake nesta versão do compilador será de acordo com as regras utilizadas na geração do código para a linguagem C. Entretanto, será a mais genérica possível de modo que não requeira um conhecimento prévio da linguagem C para compreender o que é gerado pelo Protomake. O apêndice *B* apresenta o detalhamento do código gerado pelo Protomake para a linguagem C.

4.1 Marshalling

Para cada mensagem definida pelo protocolo, um par de funções é criada para realizar as operações de *marshalling* (transformação da mensagem em uma seqüência de *bytes*) e *unmarshalling* (transformação de uma seqüência de *bytes* em uma mensagem). Para enviar uma mensagem, a operação de *marshalling* é aplicada a ela e a seqüência de *bytes* correspondente é enviada. Na recepção, a operação de *unmarshalling* é aplicada à seqüência de *bytes* e a mensagem correspondente é retornada para o programa. Durante a operação de *marshalling*, um código exclusivo de 2 (dois) *bytes* é acrescentado no início da seqüência de *bytes* e, na operação de *unmarshalling*, esse código é removido.

Para realizar o *marshalling*, as seguintes regras são aplicadas:

- O código da mensagem, de tamanho 2 *bytes*, é colocado no início da seqüência.
- Os valores dos campos do tipo *bits* são armazenados na menor quantidade de bytes que são necessários para representá-los e a quantidade de *bytes* utilizada é sempre uma potência de 2: 1 (um) *byte* para valores de até 8 (oito) *bits*, 2 (dois) *bytes* para valores entre 9 (nove) e 16 *bits (dezesseis)* e 4 (quatro) *bytes* para valores entre 17 (dezessete) e 32 (trinta e dois) *bits*.

- Os valores dos campos do tipo bytes são armazenados usando a quantidade de bytes especificados na mensagem. Caso o tamanho tenha sido especificado usando um campo previamente definido, o valor contido nesse campo previamente definido no momento de envio da mensagem é utilizado como o tamanho do campo atual.

Para realizar o unmarshalling, as seguintes regras são aplicadas:

- O código da mensagem, de tamanho 2 *bytes*, é verificado e retirado da seqüência de *bytes*, caso a mensagem seja válida.
- Os valores dos campos são extraídos levando em conta a regra de *marshalling* para os campos do tipo *bits* ou *bytes*.

4.1.1 O código de identificação e protocolos conhecidos

A ordem de chegada das mensagens para o Protomake é não-determinística. Isso possibilita a criação de protocolos mais complexos, porém introduz uma dificuldade a mais para o reconhecimento de mensagens. Essa dificuldade existe porque mensagens em ETAP podem ser enviados tanto via UDP como TCP.

UDP é um protocolo de transporte orientado a mensagens. Isso significa que, se uma mensagem de `n` *bytes* for enviada, caso ela chegue ao destino, ela será recebida por completo e de uma só vez. No caso do protocolo TCP isso pode não acontecer porque ele é orientado a *bytes*. Embora a entrega dos *bytes* seja garantida pelo TCP, não há garantia de que uma mensagem seja entregue de uma só vez. Por exemplo, se uma mensagem de 200 *bytes* é enviada, os 200 *bytes* irão chegar; no entanto, pode ser que chegue primeiro 30, depois 131 e depois 39 *bytes*, ou seja não existe uma garantia de como esta mensagem será particionada, se for o caso. A única garantia é de toda a mensagem será entregue, o que dificulta o identificação de mensagens.

Para resolver esse problema, um código de identificação de 2 *bytes* é acrescentado no início de cada mensagem enviada, resolvendo o problema de identificação. No entanto, esse código modifica a mensagem, de forma transparente para o usuário, mas impedindo que protocolos conhecidos possam ser gerados pelo compilador, pois toda mensagem terá um campo adicional que não está previsto nas descrições dos protocolos já estabelecidos.

Para eliminar esse problema, um motor de mensagens determinísticas está em desenvolvimento para ser usado com o Protomake. Desse modo, quando for usar o compilador, pode-se optar entre o motor não-determinístico (para protocolos mais complexos) e o determinístico (para protocolos não tão complexos ou já estabelecidos).

4.2 Behaviors

Quando o compilador encontra uma ação do tipo *behavior1.função(parâmetros)* e *behavior1* foi previamente definido com a palavra chave **behavior**, ele gera a interface necessária, que depende da linguagem alvo, para que essa função possa ser usada dentro do protocolo. Essa interface consiste em esqueletos das funções que devem ser preenchidos pelo programador.

Funções de um *behavior* devem sempre ser definidas com nomes diferente, pois ETAP não possui suporte a *overload* de funções. Essa decisão foi tomada porque nem toda linguagem suporta *overload,* o que dificultaria o desenvolvimento de suporte para linguagens alvo que não suportam essa característica. Se uma função de um mesmo *behavior* for usada mais de uma vez e a quantidade e/ou tipo dos parâmetros for diferente, o compilador reconhece a primeira forma como a correta e finaliza com um erro, indicando que mais de uma definição foi dada para uma mesma função.

4.3 Motor do protocolo

O motor do protocolo é o cerne do código a ser gerado pelo Protomake. Ele é composto de um *thread* de recepção e de uma máquina de estados que é construída com base na especificação feita em ETAP.

4.3.1 Thread de recepção

O *thread* de recepção é responsável pelo reconhecimento, recepção e armazenamento de todas as mensagens do protocolo. Cada mensagem recebida é colocada em uma fila de mensagens e uma notificação será enviada para todos os processos ativos, indicando que uma mensagem foi recebida. Os processos, por sua vez, verificam a mensagem e caso alguma guarda de recepção for satisfeita com ela, o processa retira essa mensagem da fila e continua com o seu processamento.

O *thread* de recepção não é gerado pelo Protomake, ele faz parte da biblioteca de *run-time* do mesmo. Apenas a função de reconhecimento de mensagens é gerada para ser

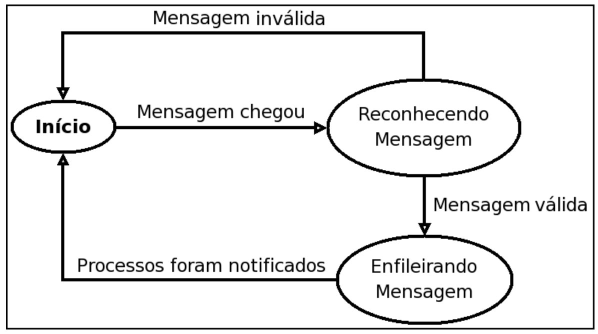

Figura 9: Máquina de estados do thread de recepção.

usada pelo *thread*. A figura 9 ilustra o funcionamento do *thread* de recepção.

4.3.2 Máquina de estados

Para cada protocolo, uma máquina de estados específica é gerada. O funcionamento da máquina, ilustrado na figura 10, no entanto, é o mesmo para qualquer tipo de protocolo, e os seguintes passos descrevem esta máquina:

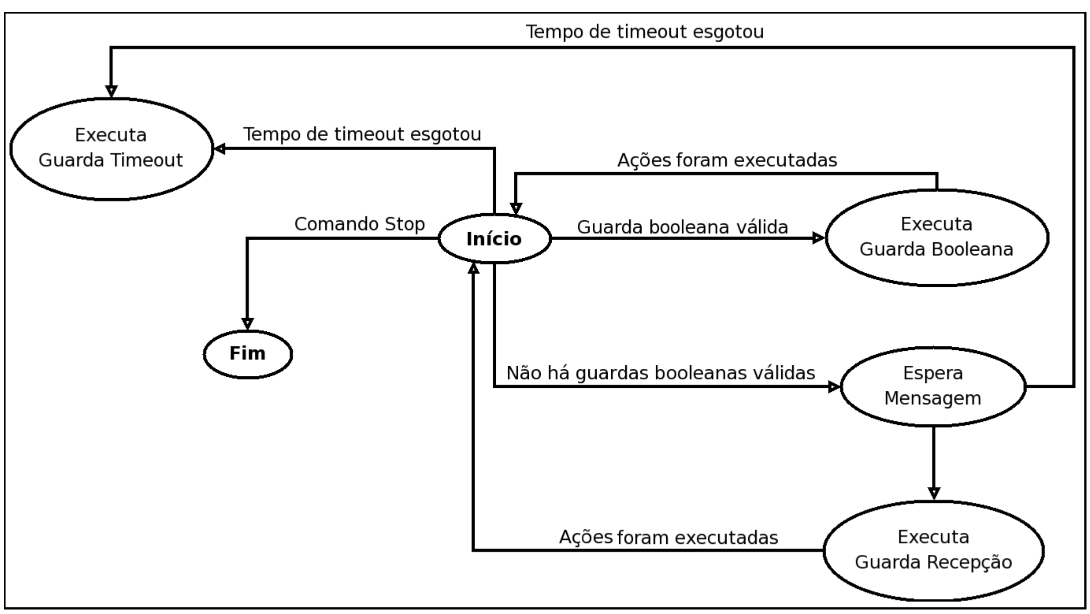

Figura 10: Máquina de estados genérica.

1. Primeiro todos as guardas booleanas são verificadas;
2. Caso exista alguma válida, as ações correspondentes são executadas e a máquina volta para o estado inicial;

3. Caso não exista uma guarda booleana válida, a máquina verifica se existe alguma mensagem na fila;

4. Caso não exista uma mensagem válida, a máquina fica parada, esperando uma nova mensagem;

5. Caso uma mensagem válida exista, as ações correspondentes são executadas e a máquina volta para o estado inicial;

6. Existe ainda a possibilidade de um *timeout* ser ativado. Nesse caso, quando o tempo do *timeout* é ultrapassado, ele será a próxima guarda a ter suas ações executadas e a máquina volta para o estado inicial.

4.4 Mapeamento de endereços

Todo endereço constante definido no protocolo deve ser especificado para que o motor do protocolo seja iniciado corretamente. Além disso, para os protocolos que usam canais de comunicação de servidores (ver 3.5.1), um endereço extra deve ser definido, que é o endereço do protocolo. Esses endereços são definidos em um arquivo de extensão .ip como ilustrado na figura 11.

```
# Exemplos de endereços constantes e de protocolo

[ Protocol address main ]
ip = 0.0.0.0
port = 0
trans = server_tcp

[ Constant address endereco ]
ip = localhost
port = 1234
trans = udp

[ Constant address endereco2 ]
ip = 192.168.0.1
port = 3565
trans = client_tcp
```

Figura 11: Exemplo de mapeamento de endereços.

Esse arquivo é gerado pelo compilador e apenas *ip* e *port* (porta) devem ser modificados. *trans* indica o método de transporte, `udp`, `server_tcp` e `client_tcp`.

Quando *trans* é `server_tcp`, ele indica que o protocolo está usando canais do tipo **client_address**. Se `server_tcp` estiver especificado no endereço de protocolo, indica que esse endereço é de um servidor TCP; quando *trans* é `client_tcp`, indica que o protocolo está usando canais do tipo **server_address**; quando *trans* é `udp`, indica que o protocolo está usando canais do tipo **address**.

4.5 Inicialização

A aplicação padrão gerada possui o código de inicialização do motor do protocolo. Em um primeiro passo, todas as variáveis relativas ao motor são inicializadas. Nesse momento, os endereços constantes são lidos e armazenados. No passo seguinte, o motor é inicializado e o protocolo passa a funcionar. Desse ponto em diante, a aplicação funciona de acordo com a descrição do protocolo, até ser finalizado.

4.6 Plugin de visualização para o Ethereal

O Protomake pode gerar um *plugin* de visualização para o *sniffer Ethereal*. Este *plugin* modifica a forma padrão de visualização das mensagens (um seqüencia opaca de bytes) da seguinte forma:

- As mensagens são exibidas com seus nomes e os nomes de seus campos como descritos na especificação;
- Os campos do tipo *bits* são exibidos como inteiros no formato hexadecimal;
- Os campos do tipo *bytes* são exibidos como um string.

A figura 12 mostra uma tela do *Ethereal* exibindo a mensagem de um protocolo não conhecido, criado através do Protomake. Já na figura 13, essa mesma mensagem é exibida utilizando o *plugin* de visualização gerado pelo Protomake.

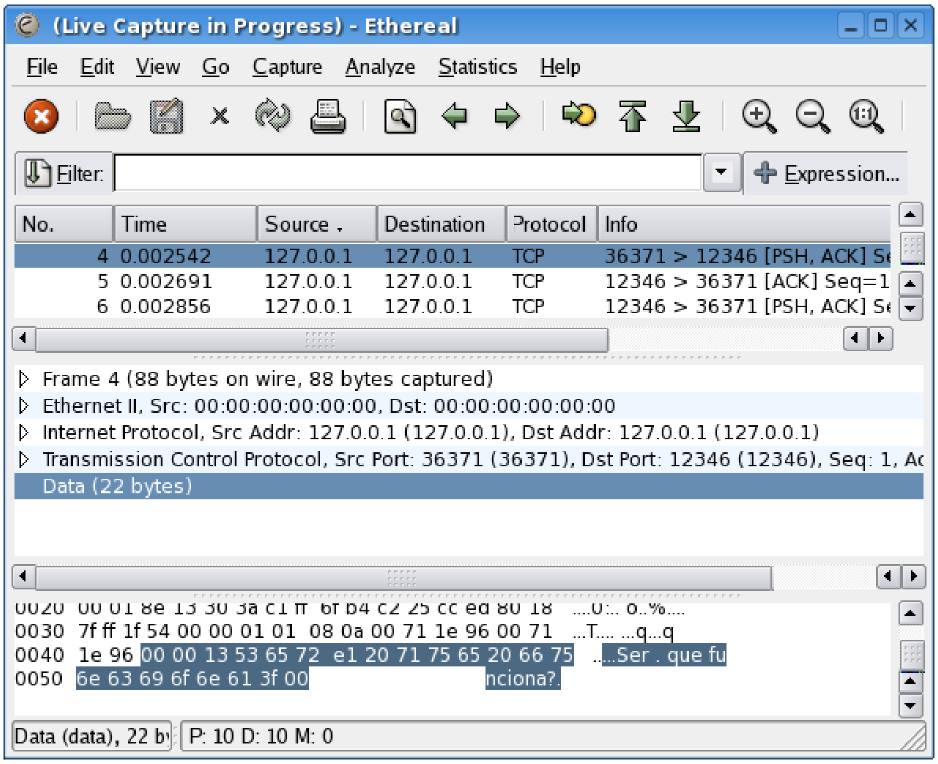

Figura 12: Ethereal sem o plugin de visualização.

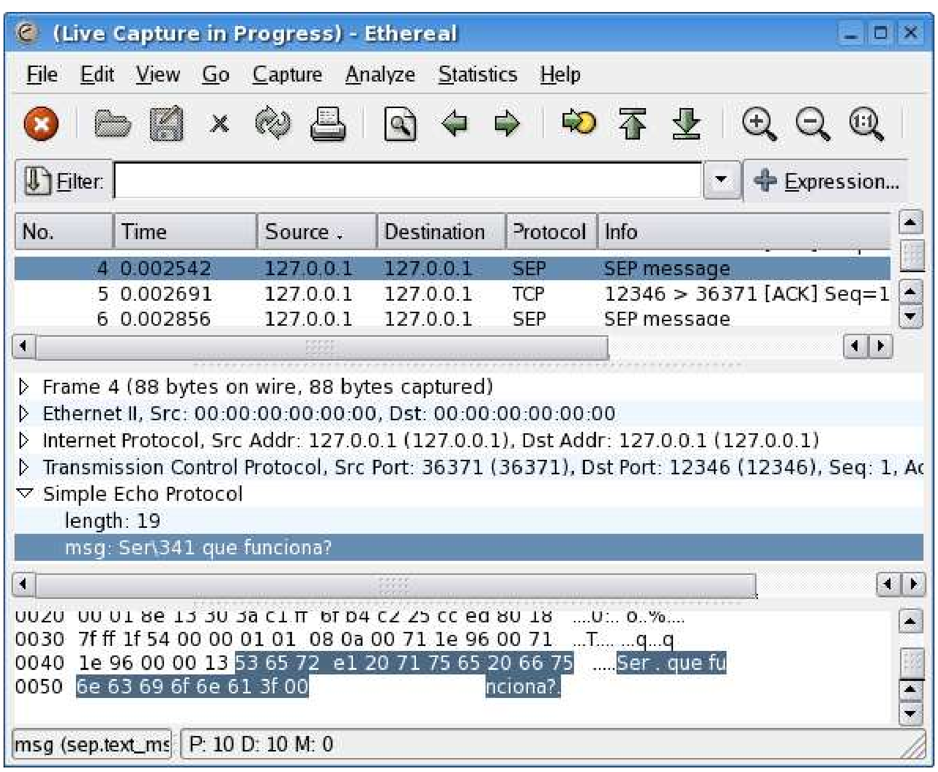

Figura 13: Ethereal com o plugin de visualização.

5 Conclusão

Este trabalho apresentou Protomake, uma ferramenta criada para facilitar o desenvolvimento de aplicações que usam protocolos de comunicação para redes de computadores. A linguagem de propósito específico ETAP foi criada como a linguagem de especificação de protocolos de Protomake.

Entre as principais vantagens de Protomake estão o nível de abstração mais alto da descrição do protocolo, sem comprometer a integração com a linguagem alvo; a possibilidade de criar protocolos complexos que enviam e recebem mensagens sem qualquer determinismo com relação a ordem das mensagens; a simplicidade de ETAP, a linguagem de descrição; a possibilidade de gerar o código do motor do protocolo e as funções de *marshalling* e *unmarshalling* para várias linguagens alvo, além de possuir uma biblioteca de *run-time* com as principais primitivas utilizadas em protocolos de comunicação (envio e recepção); e a geração de um *plugin* de visualização de mensagens para o *Ethereal*, facilitando a depuração de protocolos.

Como trabalhos futuros, modificações na linguagem serão feitas para acrescentar o tipo booleana para variávieis de processos e uma extensão do tipo *bytes* para poder especificar seqüências de bytes de tamanho variável com um byte finalizador (por exemplo, uma seqüência de caracteres que termina com o byte 0).

Além disso, serão desenvolvidas extensões para o Protomake, as quais irão disponibilizar uma interface para geração de protocolos de outras camadas, tais como transporte e enlace. É previsto o desenvolvimento de um mecanismo de *plugin* que facilite ainda mais a criação de suporte para geração de código em outras linguagens alvo. Outro ponto interessante a ser ressaltado é a construção de uma interface gráfica amigável, para tornar mais simples a interação com os usuários.

Referências

[1] Ethereal, the world's most popular network protocol analyzer, http://www.ethereal.com.

[2] Mark B. Abbot and Larry L. Peterson. A language-based approach to protocol implementation, September 1993.

[3] Ibrahim S. Abdullah and Daniel A. Menascé. Protocol specification and automatic implementation using xml and cbse. In *Proc. of Int. Conf. on Communications, Internet and Information Tech. (CIIT2003)*, pages 17,19, November 2003.

[4] A. Basu, T. von Eicken, and G. Morrisett. Promela++: A language for correct and efficient protocol construction. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, March 1998.

[5] Gerard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] T. Bolognesi and E. Brinksma. *The Formal Description Technique LOTOS*. Elsevier Science Publishers North-Holland, 1989.

[7] The International Engineering Consortium. *Specification and Description Language.*

[8] Thiago Farias, Igor Cananéa, Fábio Almeida, Rodrigo Araújo, Djamel Sadok, and Judith Kelner. Protomake: Um compilador de protocolos. In *SBRC2004 - 22o. Simpósio Brasileiro de Redes de Computadores*, volume 2, pages 881,887, May 2004.

[9] Gerard J. Holzman. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[10] E. Kohler. Prolac: A language for protocol compilation. Master's thesis, Massachusetts Institute of Technology, August 1997.

[11] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable tcp in the prolac protocol language. *SIGCOMM Comput. Commun. Rev.*, 29(4):3–13, 1999.

[12] Tommy M. McGuire. Implementing abstract protocols in c. Master's thesis, The University of Texas, December 1994.

APÊNDICE A – *Exemplos de protocolos em ETAP*

Para melhor ilustrar a utilidade da linguagem ETAP, um protocolo de *echo* é descrito utilizando a linguagem ETAP. Tanto o cliente como o servidor são especificados. Embora o protocolo de *echo* seja simples, ele ilustra a maioria das construções existentes na linguagem.

A.1 Echo

O protocolo de echo implementado consiste de uma mensagem, `msg`, com 2 (dois) campos: `tamanho`, que deve conter o tamanho da mensagem (o valor entre 1 e $2^{16} - 1$) e `mensagem`, que deve conter a mensagem propriamente dita. O protocolo de transporte usado é o UDP e o servidor sempre espera uma mensagem de reconhecimento do cliente, `ack`, para confirmar o recebimento. Caso a mensagem não chegue dentro do *timeout* especificado, a mensagem é retransmitida para o cliente.

A.1.1 Código do servidor

```
# Mensagem do protocolo de echo
message msg
begin
    tamanho : 16 bits
    mensagem : tamanho bytes
end

# Mensagem de reconhecimento
message ack
begin
    ack : 1 bits
end

# Processo do protocolo echo
```

```
process main
var cliente : address
begin
    rcv msg from cliente -> send msg to cliente
                            act ackTimeout in 10000
    rcv ack from cliente -> if ack.ack = 1 ->
                                cancel ackTimeout
                            fi
    [] timeout ackTimeout -> send msg to cliente
                             act ackTimeout in 10000
end
```

A.1.2 Código do cliente

```
# Mensagem do protocolo de echo
message msg
begin
    tamanho : 16 bits
    mensagem : tamanho bytes
end

# Mensagem de reconhecimento
message ack
begin
    ack : 1 bits
end

behavior consoleIO

# Processo do protocolo echo
process main
const servidor : address
var first : integer = 1
begin
    first = 1 -> first := 0
                 consoleIO.readMessage(msg)
                 send msg to servidor
                 act response in 10000
    [] rcv msg from servidor -> cancel response
                                consoleIO.writeMessage(msg)
                                ack.ack := 1
                                send ack to servidor
                                stop
    [] timeout response -> send msg to servidor
                           act response in 10000
end
```

APÊNDICE B – Código gerado em C

Todo o código gerado pelo Protomake a partir da descrição do cliente de *echo*, apresentado no apêndice *A*, é aqui descrito. Além dele, o *thread* de recepção também é descrito, pois ele é sempre o mesmo para qualquer protocolo criado.

B.1 Estruturas de dados

Mensagem `msg`:

```
typedef struct message_msg {
    uint16_t tamanho;
    byte *mensagem;
} message_msg;
```

Mensagem `ack`:

```
typedef struct message_ack {
    byte ack;
} message_ack;
```

B.2 Marshalling e unmarshalling

Função de marshalling da mensagem `msg`:

```
byte *message_msg_marshall(const message_msg *msg_msg,
                           byte *buffer,
                           int *length) {
    uint16_t code = htons(0);
    uint32_t offset = 0;
```

```
    uint16_t temp16 = 0;
    uint32_t temp32 = 0;
    byte *buf;

    *length = sizeof(uint16_t) + 2 + msg_msg->tamanho;
    buf = (byte *) malloc((*length) * sizeof(byte));
    memcpy(buf, &code, 2);  offset = 2;
    temp16 = htons(msg_msg->tamanho);
    memcpy((buf + offset), &temp16, 2);
    offset += 2;

    memcpy((buf + offset), msg_msg->mensagem, msg_msg->tamanho);
    offset += msg_msg->tamanho;

    if (buffer != NULL) {
        memmove(buffer, buf, offset);
        free(buf);
        buf = NULL;
    }
    return buf;
}
```

Função de unmarshalling da mensagem `msg`:

```
message_msg *message_msg_unmarshall(const byte *buffer,
                                    message_msg *msg_msg) {
    message_msg *msg = (message_msg *) malloc(sizeof(message_msg));
    uint32_t offset = 2;

    memcpy(&(msg->tamanho), (buffer + offset), 2);
    offset += 2;

    msg->tamanho = ntohs(msg->tamanho);
    msg->mensagem = (byte*) malloc(sizeof(byte) * msg->tamanho);
    memcpy(msg->mensagem, (buffer + offset), msg->tamanho);
    offset += msg->tamanho;

    if (msg_msg != NULL) {
        memmove(msg_msg, msg, sizeof(message_msg));
        free(msg);
        msg = NULL;
    }
    return msg;
}
```

Função de marshalling da mensagem `ack`:

```
byte *message_ack_marshall(const message_ack *msg_ack,
                           byte *buffer,
                           int *length) {
    uint16_t code = htons(1);
    uint32_t offset = 0;
    uint16_t temp16 = 0;
    uint32_t temp32 = 0;
    byte *buf;

    *length = sizeof(uint16_t) + 1;
    buf = (byte *) malloc((*length) * sizeof(byte));

    memcpy(buf, &code, 2);
    offset = 2;

    memcpy((buf + offset), &(msg_ack->ack), 1);
    offset += 1;

    if (buffer != NULL) {
        memmove(buffer, buf, offset);
        free(buf);
        buf = NULL;
    }
    return buf;
}
```

Função de unmarshalling da mensagem `ack`:

```
message_ack *message_ack_unmarshall(const byte *buffer,
                                    message_ack *msg_ack) {
    message_ack *msg = (message_ack *) malloc(sizeof(message_ack));
    uint32_t offset = 2;

    memcpy(&(msg->ack), (buffer + offset), 1);
    offset += 1;

    if (msg_ack != NULL) {
        memmove(msg_ack, msg, sizeof(message_ack));
        free(msg);
        msg = NULL;
    }
    return msg;
}
```

B.3 Esqueletos das funções de behavior

Esqueleto da função `consoleIO.readMessage`:

```
void consoleIO_readMessage(message_msg *param1) { }
```

Esqueleto da função `consoleIO.writeMessage`:

```
void consoleIO_writeMessage(message_msg *param1) { }
```

B.4 Máquina do protocolo

```
void thread_process_main(void *param) {
    message_msg *pMsg =
        (message_msg *) malloc(sizeof(message_msg));
    message_ack *pAck =
        (message_ack *) malloc(sizeof(message_ack));
    int end = 0;
    GenericMessage *message = NULL;
    int first = 1;

    pMsg->tamanho = 0;
    pMsg->mensagem = NULL;
    pAck->ack = 0;

    while(!end) {
        int estado = 1;
        if ((first == 1)) {
            estado = 0;
        }
        switch(estado) {
            case 0:
                first = 0;
                {
                    message_msg *param1 = pMsg;

                    consoleIO_readMessage(param1);
                }
                {
                    int message_msg_length;
                    PMSocket *sock;

                    byte *buf =
                        message_msg_marshall(pMsg, NULL,
```

```
                                     &message_msg_length);

        if (socketHash == NULL ||
            (sock = pmhashtable_get(socketHash, "servidor")) == NULL)
            sock = message->sock;

        if (pmsocket_send(sock, buf, message_msg_length) != 0) {
           exit(1);
        }
    }
    break;

case 1:
{
    int codes[1] = {0};
    if (message != NULL) {
        if (message->msg != NULL)
            free(message->msg);
        free(message);
        message = NULL;
    }

    message = pmqueue_dequeue(messageQueue, 1, codes, NULL);

    if (message) {
        switch(message->code) {
            case 0: {
                pMsg = (message_msg*) message->msg;

                {
                    message_msg *param1 = pMsg;

                    consoleIO_writeMessage(param1);
                }
                pAck->ack = 1;
                {
                    int message_ack_length;
                    PMSocket *sock;

                    byte *buf =
                        message_ack_marshall(pAck, NULL,
                                             &message_ack_length);

                    if (socketHash == NULL ||
                        (sock = pmhashtable_get(socketHash, "servidor")) == NULL)
                        sock = message->sock;

                    if (pmsocket_send(sock, buf, message_ack_length) != 0) {
                        exit(1);
```

```
                        }
                    }
                    end = 1;
                    break;
                }
            }
        }
    }
    break;
    }
  }
}
```

B.5 Função decodificadora

```
GenericMessage *decodeMessage(PMSocket *sock) {
    GenericMessage *msg = (GenericMessage*) malloc(sizeof(GenericMessage));
    byte validMsg = 1;
    uint16_t code;
    uint32_t size;
    byte *udpBuffer;
    PMSocket *sender = (PMSocket *) malloc(sizeof(PMSocket));

    if (!sock)
        return NULL;

    if (sock->trans != UDP) {
        size = 2;
        pmsocket_recv(sock, (byte*) &code, &size, sender);
        msg->sock = sock;
    }
    else if (sock->trans == UDP) {
        size = 0xFFFF;
        udpBuffer = (byte *) malloc(sizeof(byte) * size);
        pmsocket_recv(sock, udpBuffer, &size, sender);
        msg->sock = sender;
        memcpy(&code, udpBuffer, 2);
    }

    msg->code = ntohs(code);
    switch (msg->code) {
        case 0: {
            message_msg *msgPtr = (message_msg*) malloc(sizeof(message_msg));
            if (sock->trans != UDP) {
                size = 2;
```

```
            pmsocket_recv(sock, (byte*) &(msgPtr->tamanho), &size, sender);
            size = msgPtr->tamanho;
            msgPtr->mensagem = (byte *) malloc(sizeof(byte) * size);
            pmsocket_recv(sock, msgPtr->mensagem, &size, sender);
        }
        else if (sock->trans == UDP) {
            uint32_t offset = 2;
            size = 2;
            memcpy(&(msgPtr->tamanho), udpBuffer + offset, size);
            offset += size;

            size = msgPtr->tamanho;
            msgPtr->mensagem = (byte *) malloc(sizeof(byte) * size);
            memcpy(msgPtr->mensagem, udpBuffer + offset, size);
        }
        msg->msg = msgPtr;
        free(udpBuffer);
        break;
    }

    case 1: {
        message_ack *msgPtr = (message_ack*) malloc(sizeof(message_ack));
        if (sock->trans != UDP) {
            size = 1;
            pmsocket_recv(sock, (byte*) &(msgPtr->ack), &size, sender);
        }
        else if (sock->trans == UDP) {
            uint32_t offset = 2;
            size = 1;
            memcpy(&(msgPtr->ack), udpBuffer + offset, size);
        }
        msg->msg = msgPtr;
        free(udpBuffer);
        break;
    }
  }
  if (!validMsg) {
    free(msg);
    msg = NULL;
  }
  return msg;
}
```

B.6 Thread de recepção

```
void *threadreceiver(void *arg) {
```

```
pmsocketlist_iterator *iter;
GenericMessage *message = NULL;
struct pollfd *fdSet =
    (struct pollfd *) malloc(sizeof(struct pollfd) * MAXFDSET);
int end = 0;
PMSocket **socket_poll =
    (PMSocket **) malloc(sizeof(PMSocket*) * MAXFDSET);
int last = 0;

pthread_testcancel();
pmsocketlist_lock(socketList);

iter = socketList->head;
while (iter != NULL) {
    fdSet[last].fd = iter->socket->socketFD;
    fdSet[last].events = POLLIN | POLLPRI;
    fdSet[last].revents = 0;
    socket_poll[last] = iter->socket;
    iter = iter->next;
    last++;
}
pmsocketlist_unlock(socketList);

pthread_testcancel();
while (!end) {
    int totalFds;
    int i;
    char info[256];

    totalFds = poll(fdSet, last, -1);

    pthread_testcancel();
    for (i = 0; i < last && totalFds; i++) {

        if (fdSet[i].revents != 0) {
            PMSocket *sock = socket_poll[i];

            if (fdSet[i].revents & (POLLERR | POLLHUP)) {
                pmsocketlist_remove(socketList, sock);

                if (i < last - 1) {
                    fdSet[i].fd = fdSet[last-1].fd;
                    fdSet[i].revents = fdSet[last-1].revents;
                    socket_poll[i] = socket_poll[last-1];
                     i--;
                }
                last--;
            }
```

```
else if (fdSet[i].revents) {
    if (sock->trans != UDP) {

        if (pmsocket_wasClosed(sock)) {
            pmsocketlist_remove(socketList, sock);

            if (i < last - 1) {
                fdSet[i].fd = fdSet[last-1].fd;
                fdSet[i].revents = fdSet[last-1].revents;
                socket_poll[i] = socket_poll[last-1];
                i--;
            }
            last--;
        }
        else {
            if (!sock->client) {
                sock = pmsocket_accept(sock);

                sock->client = 1;
                sock->connected = 1;
                sock->var = 1;

                pmsocketlist_add(socketList, sock, NULL);
                if (last < MAXFDSET) {
                    fdSet[last].fd = sock->socketFD;
                    fdSet[last].events = POLLIN | POLLPRI;
                    fdSet[last].revents = 0;
                    socket_poll[last] = sock;
                    last++;
                }
            }

            if (sock->var) {
                 message = decodeMessage(sock);
                 pmqueue_enqueue(messageQueue, message);
            }
            else {
                message = decodeMessage(sock);
                pmqueue_enqueue(messageQueue, message);
            }
        }
    }
    else {
        message = decodeMessage(sock);
        pmqueue_enqueue(messageQueue, message);
    }
}
```

```
                totalFds--;
            }
        }
        pthread_testcancel();
    }

    free(fdSet);
    free(socket_poll);

    return NULL;
}
```