4
Estudos de Caso

Neste capítulo descreveremos as experiências obtidas nas verificações dos protocolos RDP e DSR. Anteriormente à definição da arquitetura, ambos foram especificados em Spin e uma análise comparativa das vantagens e desvantagens da adoção da arquitetura são relatadas ao longo do capítulo. Um dos fatores que serão discutidos é o tamanho (número de linhas, comandos, etc) das especificações, que têm grande importância, considerando-se a proposta de uma arquitetura mais abstrata para a descrição de protocolos.

4.1
Protocolos Analisados

Nesta seção detalharemos o processo de especificação e verificação dos protocolos RDP e DSR. Dividiremos esta análise em descrição informal do protocolo, suposições assumidas na modelagem, especificação em LEP e verificação formal.

4.1.1
RDP

RDP (Result Delivery Protocol) (ESO00) otimiza e garante a entrega de respostas de hosts fixos para clientes móveis, mesmo com desconexões intermitentes e migrações de hosts. Nesta subseção descreveremos brevemente uma versão inicial deste protocolo, no qual um cliente móvel cria e mantém proxies em diferentes estações base, cada qual relacionada à um dado pedido. RDP tem algumas características que o difere do IP-móvel (provavelmente o protocolo mais conhecido para redes estruturadas móveis): envio confiável de mensagens devido às retransmissões; proxies criados dinamicamente sob demanda ao invés dos *home-agents* (proxies fixos no IP-móvel), e; interações no formato pedido-resposta. Para cada pedido de cliente o protocolo cria um proxy na estação base corrente do cliente. Cada proxy guarda informações necessárias para rastrear as migrações dos clientes móveis pelas estações base e é responsável por receber, armazenar e encaminhar respostas da localização corrente para o cliente móvel. Proxies são criados para o primeiro pedido

PUC-Rio - Certificação Digital Nº 0115606/CA

na estação base e são reutilizados para outros pedidos enquanto existirem pedidos pendentes. Proxies só são removidos quando não há mais mensagem para responder.

As estações base (MSS) são parte da infra-estrutura sem fio e tem papel fundamental no *handoff* (sub-protocolo que lida com a parte de migração e delegação de responsabilidades entre MSS's). Quando um cliente móvel troca de célula, ele envia uma mensagem *greet* para se anunciar a uma nova MSS. Deste ponto em diante as MSS anterior e atual trocam mensagens de forma que a nova MSS possa ser responsável pelo cliente recém chegado. Uma mensagem do tipo *greet* também é enviada quando um cliente se reconecta após um período de desconexão.

A MSC (Message Sequence Chart - Diagrama de Sequência de Mensagens) descrita na figura 4.1 mostra um cenário simples onde o sub-protocolo de *handoff* é ativado.

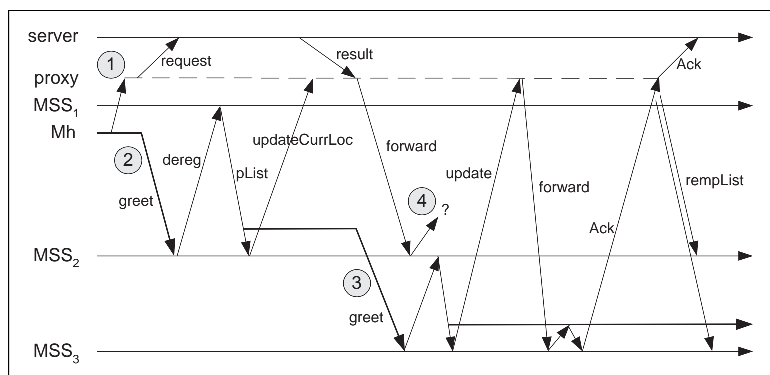

Figura 4.1: MSC de um cenário do RDP

Linhas horizontais são entidades e linhas orientadas são mensagens. No rótulo 1, o cliente móvel (Mh) envia um pedido para a estação base atual (Mss1). Esta estação cria um proxy (linhas pontilhadas) que é responsável pelos pedidos. No rótulo 2, Mh se anuncia (mensagem *greet*) para Mss2, a qual troca mensagens com Mss1 (*dereg* e *plist*) para se tornar responsável pelo Mh (protocolo *handoff*). Então, Mss2 envia uma mensagem ao proxy (*updateCurrLoc*) de forma a atualizar a localização do cliente. O rótulo 3 indica uma outra migração de Mh. Entretanto, antes do término da migração, Mss2 recebe a resposta do pedido do Mh (mensagem *forward*). O ponto de interrogação no rótulo 4 indica que o Mh já deixou a célula do Mss2. Após o término da nova migração, o proxy é atualizado e retransmite a mensagem. Ao receber a mensagem *ack* proveniente da estação base, o proxy envia um *ack* ao servidor e uma mensagem *rempList* para todas as estações base visitadas por Mh, significando que as referências para Mh deverão ser removidas. Se o Mh não fizer novos pedidos, o proxy é removido.

PUC-Rio - Certificação Digital Nº 0115606/CA

Em RDP, a confiança da entrega das mensagens permite uma possível replicação de mensagens, uma vez que os proxies retransmitem as mensagens pendentes sempre que eles recebem uma nova mensagem de atualização de localização (mensagem *updateCurrLoc*). Como a criação de proxies depende da movimentação do cliente sobre a rede, proxies podem ser criados nas diferentes estações base onde o cliente fez pedidos. Com isso, a ordem do recebimento de mensagens não pode ser garantida, já que uma estação base pode ser mais rápida do que outra.

Suposições

Na modelagem do protocolo RDP, a herança das suposições deste protocolo é inevitável já que queremos especificar de forma fiel o comportamento deste protocolo. Resumidamente, as suposições deste protocolo são (a descrição completa encontra-se em (ESO00)):

1. As estações base (MSSs) não falham. A comunicação entre elas e o servidor é confiável e a ordem de entrega das mensagens é indefinida;

2. A qualquer momento, cada agente móvel (mh) está associado a uma única estação base;

3. Se um mh está ativo, este precisa enviar uma mensagem *Ack* (*Acknowledgment*) para todas as mensagens recebidas de sua estação base; se estiver inativo, não deve responder nenhuma mensagem;

4. Um agente móvel é capaz de detectar se uma mensagem enviada pela sua estação base é nova ou é uma retransmissão;

5. Um agente móvel somente deixa o sistema (torna-se inativo) após enviar todas as mensagens *Ack* referentes às mensagens que recebeu;

6. Foi abstraída a maneira como um agente móvel percebe que está entrando ou saindo da célula de alguma estação base, supondo que isto é resolvido pela tecnologia sem fio adotada.

Suposições também poderiam ser impostas pela ferramenta de verificação adotada. Entretanto, no nosso entender, isto caracterizaria uma incompatibilidade entre o que se deseja verificar e a ferramenta de verificação utilizada. Ou seja, tanto a arquitetura quanto as ferramentas de verificação adotadas por ela não podem impor qualquer tipo de restrição que nos force a definir uma nova suposição.

PUC-Rio - Certificação Digital Nº 0115606/CA

Especificação

Na modelagem do protocolo RDP, cada elemento do protocolo é um módulo em LEP. Para simplificarmos a especificação, abstraímos os proxies num único módulo, que guardará referências de todos os agentes móveis criados. A configuração inicial da rede é fornecida de forma explícita, interligando as estações base, o servidor e o proxy de formal total, e deixando a estação móvel inicialmente numa das estações base, como podemos ver na figura 4.2. Observe que não foi necessário adicionar a conexão do *mh*, por exemplo, já que a topologia foi declarada como tendo conexões não-direcionadas (*undirected*). Ou seja, ao declararmos o agente *mh* na vizinhança do elemento *mss[1]*, atualizamos as vizinhanças de forma mútua. Estas conexões são inseridas no pronome *neighbours* de cada um destes elementos de forma transparente.

```
topology is
{server - {mss:1..2},
 mss[1] - {mh, mss[2]},
 mss[2] - {},
 mh - {},
 proxy - {server, mss:1..2}} undirected;
```

Figura 4.2: Topologia inicial de RDP em LEP

Uma outra maneira de prover a configuração inicial da topologia seria inserirmos as conexões explicitamente na especificação dos módulos, atualizando os pronomes *neighbours* de cada elemento na transição inicial (*init*) de seus módulos. Com isso, para inicializarmos a estação base de um agente móvel teríamos: *neighbours = any(mss)*. Ou seja, a estação base de um agente móvel seria escolhida aleatoriamente dentre as estações existentes, o que torna a especificação mais geral.

De forma a analisarmos o comportamento do protocolo no cenário de *handoff*, o módulo *mh* (figura 4.3), que modela uma estação móvel, inicia fazendo um requerimento (mensagem *request*) para sua estação base e, em seguida, sinaliza a intenção de movimento, ou troca de célula (mensagem *greet*).

```
module mh
        pronoun mymss, oldmss;

        init -> neighbours!request;
                oldmss = neighbours;
                mymss = anyother(oldmss);
                mymss!greet(oldmss);

        this?response ->
                sender!mh_ack;
endmodule
```

Figura 4.3: Estação móvel do RDP em LEP

PUC-Rio - Certificação Digital Nº 0115606/CA

Modelamos as estações base através do módulo apresentado na figura 4.4 para o atendimento das solicitações feitas pelas estações móveis e a comunicação com outras estações (por exemplo, durante o *handoff*).

```
module mss
        pronoun partner;
        this?request ->
                if neighbours.contains(sender) then
                        proxy!request(sender);
                endif
        this?greet(partner) ->
                partner!dereg(sender);
        this?dereg(partner) ->
                neighbours.del(partner);
                sender!plist(partner);
        this?plist(partner) ->
                neighbours.add(partner);
                proxy!updateLocation(partner, this);
        this?forward(partner) ->
                if neighbours.contains(partner) then
                        partner!response;
                endif
        this?mh_ack ->
                proxy!ack(sender);
endmodule
```

Figura 4.4: Estação base do RDP em LEP

Para cada pedido feito por um agente móvel, a estação base primeiramente testa se o agente ainda é de responsabilidade desta estação, ou seja, se o agente ainda está em sua vizinhança. Estando, uma mensagem é enviada ao proxy contendo o agente móvel que fez a requisição. O módulo que especifica o proxy é dado na figura 4.5.

Ao receber um pedido (mensagem *request*), o proxy registra a estação móvel que originou o pedido e sua estação base. Após, o proxy encaminha o pedido para o servidor. Este, descrito na figura 4.6, recebe a mensagem, supostamente realiza algum processamento e depois retorna o resultado do processamento para o proxy, que reencaminhará para a estação base que seja correntemente responsável pelo agente móvel

Além do tratamento de uma requisição, a especificação dada também modela o *handoff*, que se inicia quando o agente móvel envia uma mensagem *greet* para a estação responsável pela localização para onde o agente se moveu (figura 4.3). A estação base que recebe a mensagem inicia a comunicação com a estação originária do agente, de forma a obter as informações referentes a este. Após, a estação insere o agente em sua vizinhança e atualiza o proxy informando que ela é a nova estação responsável pelo agente móvel. Ao receber

PUC-Rio - Certificação Digital Nº 0115606/CA

```
module proxy
        pronoun partner, station[], newst;
        int pending[], val

        init -> pending[] = 0;

        this?request(partner) ->
                station[partner] = sender;
                server!request(partner);

        this?result(partner) ->
                station[partner]!forward(partner);
                pending[partner]++;

        this?ack(partner) -> pending[partner]--;

        this?updateLocation(partner, newst) ->
                station[partner] = newst;
                val = pending[partner];
                while val>0 do
                        station[partner]!forward(partner);
                        val--;
                endwhile
end module
```

Figura 4.5: Abstração dos proxies do RDP em LEP

```
module server
          pronoun host;
          this?request(host) ->
                    sender!result(host);
endmodule
```

Figura 4.6: Módulo servidor do RDP em LEP

a atualização, o proxy (figura 4.5) atualiza o registro do agente móvel e reenvia para a nova estação todas as mensagens pendentes do agente.

Verificação

Na verificação do protocolo RDP estamos inicialmente interessados em analisar a garantia de entrega das mensagens, que é uma característica importante se comparada ao protocolo IP-móvel. Para tal, especificamos a seguinte propriedade (padrão *Response* de (DAC98)):

$$[] \text{ (p!request} \rightarrow <> \text{p?response)}$$

, onde *request* é a requisição do agente móvel *p* e *response* é a mensagem enviada pela estação base como resposta da requisição. Como esperado, a propriedade é verificada para a especificação apresentada. Entretanto, se ao invés de enviarmos as mensagens *request* e *greet* sequencialmente uma única vez(figura 4.3) enviarmos de forma não-determinística (figura 4.7), a propriedade anterior não se verifica.

A propriedade não é verificada pois, dado o não-determinismo, um agente móvel pode fazer uma requisição numa estação base e, incessantemente, se

PUC-Rio - Certificação Digital Nº 0115606/CA

```
module mh
        pronoun mymss, oldmss;

        true -> neighbours!request;

        true -> oldmss = neighbours;
               mymss = anyother(oldmss);
               mymss!greet(oldmss);

        this?pre_mh_ack ->
                 sender!mh_ack;
endmodule
```

Figura 4.7: Estação móvel do RDP com inicialização não-determinística

deslocar de forma tão frequente que este não tenha como receber a resposta de sua requisição. O contra-exemplo retornado pela arquitetura é:

$$\text{p!request(neighbours)} \Rightarrow \text{p!greet}(\text{mymss}_1) \Rightarrow \text{p!greet}(\text{mymss}_2)$$

, onde a repetição da mensagem *mh!greet(mymss)* indica que em algum momento esta mensagem poderia ser enviada indefinidamente. Os índices distintos no argumento *mymss* (mymss_1 e mymss_2) indicam que não estes não precisam ser necessariamente iguais.

Outra propriedade que pode ser analisada é a ordem de entrega de mensagens no RDP. Para analisá-la adicionamos um parâmetro à mensagem *request* que a rotulará. Na especificação, o parâmetro é passado de forma transparente para todas as mensagens que tratam a requisição até a entrega da resposta ao agente móvel. Com isso, basta verificar a ordem para dois (2) rótulos distintos. Assim, especificamos a seguinte propriedade:

$$[]\ (\text{p!request(1) U p!request(2)}) \rightarrow (\text{p?response(1) U p?response(2)})$$

, a qual especifica que se duas (2) mensagens rotuladas 1 e 2 são enviadas nesta ordem, implica que suas respostas serão recebidas nesta mesma ordem. Esta propriedade não se verifica, já que não há qualquer porção da especificação que force o sequenciamento das mensagens. O contra-exemplo recuperado no nível de LEP é:

$$\boxed{\text{p!request(1)} \Rightarrow \text{p!request(2)} \Rightarrow \text{p?response(2)}}$$

PUC-Rio - Certificação Digital Nº 0115606/CA

4.1.2 DSR

DSR (Dynamic Source Routing) (JMH04) (JMB01) é um protocolo simples e eficiente para redes ad-hoc. Cada pacote enviado através da rede contém os identificadores dos nós que são visitados pelo pacote, ou seja, a rota percorrida. Cada nó mantém uma cache de rotas, as quais são aprendidas através dos pacotes enviados por estes nós.

O protocolo DSR é composto de 2 sub-protocolos: um para a descoberta de rotas e outro para a manutenção destas. Quando um nó deseja enviar um pacote para outro, este procura por uma rota para o destino em sua cache. Se o nó encontra a rota, o pacote é enviado para o primeiro nó da rota e assim sucessivamente até que o pacote alcance o destino. Se não a encontra, o mecanismo para descoberta de rotas é ativado. Essa mesma ativação ocorre quando há um erro no roteamento, o qual pode ter sido causado por movimentação ou desativação dos nós.

O sub-protocolo para descoberta de rotas está descrito na figura 4.8. Quando um nó A necessita de uma rota, este envia um pedido de rota para seus vizinhos (nós que são alcançados por este). Os vizinhos fazem o mesmo até que o destino seja alcançado (*flooding*). A descoberta é enviada de volta para o nó A que então pode enviar a mensagem. Uma constante (id=1) rotula o pedido de rota, permitindo ao nó E aceitar somente o primeiro pedido, descartando os seguintes que possam ocorrer.

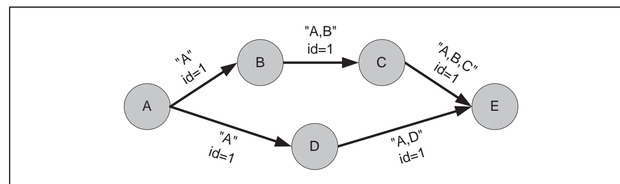

Figura 4.8: Mecanismo para a descoberta de rotas

O sub-protocolo para a manutenção de rotas valida as rotas armazenadas na cache. Se um nó detecta um erro na transmissão de um pacote, este envia um pacote sinalizando erro de volta para o nó que requisitou a mensagem, indicando qual ligação que foi quebrada. Quando um nó recebe uma mensagem de erro, este atualiza sua cache removendo todas as sub-rotas que ocorrem depois da ligação quebrada.

Várias extensões e otimizações para o protocolo DSR tem sido propostas (JMH04). Dentre elas, destacamos:

Extensões para pedidos de rota

PUC-Rio - Certificação Digital Nº 0115606/CA

– rotas podem ser reversíveis (origem para destino e vice-versa) ou não;
– os pedidos de requisição de rota podem ser atendidos por um nó através da busca em sua própria cache. Não encontrando, o pedido é encaminhado normalmente para a vizinhança do nó;
– o número de encaminhamentos dos pedidos de requisição de rota podem ser limitados.

Extensões para manutenção de rotas

– no encaminhamento de um pacote, caso o nó detecte que a ligação com o nó seguinte da sequência está quebrada, havendo caminho para o destino na cache, o nó substitui o caminho quebrado com o de sua cache e encaminha o pacote;
– no encaminhamento, quando um nó detecta que a ligação com o nó seguinte está quebrada, o mesmo tratamento dado a este pacote deve ser dados a todos os pacotes que estejam enfileirados neste nó esperando uma nova rota para encaminhamento;
– no recebimento de uma mensagem de erro, esta é repassada por *piggy-backing* no próximo pedido de requisição de rota.

Na tese nos detemos a tratar da versão básica do protocolo com alguma variações, mas que é suficiente para demonstrar a utilização da linguagem, e consequentemente da arquitetura. Além disso, não vislumbramos maiores dificuldades na especificação das extensões se compararmos com a especificação da versão básica.

PUC-Rio - Certificação Digital Nº 0115606/CA

Suposições

A especificação do protocolo DSR está baseada nas seguintes suposições:

1. Todos os nós que fazem parte da rede participam do protocolo de forma completa;
2. Pacotes podem ser perdidos ou corrompidos. No recebimento de um pacote, um nó consegue detectar que o pacote está corrompido e descartá-lo;
3. Não há interferência no recebimento de mensagens. Ou seja, um nó nunca recebe duas (2) mensagens simultaneamente, o que poderia acarretar a perda de um delas;
4. Os nós na rede ad hoc podem se deslocar a qualquer momento sem qualquer tipo de aviso prévio;
5. Os nós na rede são distinguíveis.

Especificação

Na modelagem do protocolo DSR, definimos um módulo em LEP que descreverá o comportamento de um nó da rede neste protocolo. Inicialmente, a configuração inicial da rede é fornecida de maneira implícita, através da macro *Arbitrary(X)* que, não-deterministicamente, criará as conexões iniciais da rede de tamanho X. O parâmetro *undirected* indica que as arestas na rede não são orientadas, ou seja, as rotas são reversíveis. O parâmetro *dynamic* indica que a rede terá um comportamento dinâmico gerado automaticamente e de forma transparente.

topology is Arbitrary(7) undirected dynamic;

Na primeira parte da definição do módulo (figura 4.9), declaramos suas variáveis locais e iniciamos escolhendo aleatoriamente um destino para uma mensagem a ser enviada. As variáveis *pacote* e *aux* são utilizadas para armazenar os pacotes a serem retransmitidos pelo nó. Estes pacotes são compostos de uma sequência (a rota) e de um valor inteiro (rótulo que identifica a rota). A variável *cache* armazena as rotas aprendidas pelo nó. A variável *ordem* é um contador que armazena o número de requisições iniciadas por este nó. Como dissemos anteriormente, ainda não temos em LEP comandos referentes a especificações em tempo-real. Ou seja, não temos como implementar o *timeout* no envio de requisições em LEP. De forma a simular este comportamento no envio de uma requisição, fazemos com que a *ordem* da mensagem possa ser a mesma que a anterior, simulando um reenvio, ou criamos uma nova *ordem* (*ordem={ordem, ordem+1};*).

```
module hop
        (seq:hop)#int pacote, aux;
        set:seq:hop cache;
        int ordem=0;

        true ->  pacote#1.first = this;
                 pacote#1.last = anyother(this);
                 ordem = { ordem, ordem+1 };
                 pacote#2 = ordem;
                 if cache.contains(pacote#1.last) then
                         aux = cache.range(pacote#1.first, pacote#1.last);
                         aux.next!comm(aux);
                 else
                         neighbours!rr(pacote);
                 endif
```

Figura 4.9: Declaração das variáveis locais e da mensagem que inicia o protocolo DSR

Após gerarmos aleatoriamente o destino para o envio da mensagem (*pacote#1.last = anyother(this);*), buscamos na cache uma rota que contenha

PUC-Rio - Certificação Digital Nº 0115606/CA

o destino gerado. Em caso de sucesso, a rota armazenada é utilizada para o envio da mensagem (mensagem *comm*). Caso o destino não seja encontrado, ativamos o sub-protocolo de descobrimento de rotas (mensagem *rr*).

```
this?rr (pacote) ->
        if this == pacote#1.last then
                pacote#1.add(this);
                pacote.previous!unicast(pacote);
        else
                if not(pacote#1.contains(this)) then
                        pacote#1.add(this);
                        neighbours!rr(pacote);
                endif
        endif
```

Figura 4.10: Tratamento da mensagem de descoberta de rota

No primeiro passo do sub-protocolo de descobrimento de rotas (figura 4.10), testamos se o nó que recebeu a requisição de uma rota (mensagem *rr*) é o nó destino. Adicionamos este nó na rota corrente e a reencaminhamos para o penúltimo nó inserido na rota (*pacote#1.previous!unicast*). Este reencaminhamento direto só é possível pois estamos supondo que a rota é reversível e que a rota não se quebra até que o pacote chegue à origem da requisição. Entretanto, se o nó não é o destino da rota e ainda não pertence à ela, este é adicionado à rota e a reenviamos através da mensagem *rr* para os vizinhos deste nó.

PUC-Rio - Certificação Digital Nº 0115606/CA

```
this?unicast (pacote) ->
        cache.add(pacote);
        if this == pacote#1.first then
                pacote#1.next!comm(pacote);
        else
                pacote#1.previous!unicast(pacote);
        endif
```

Figura 4.11: Tratamento da mensagem que retorna a rota obtida para o nó origem

Na figura 4.11 temos o tratamento da mensagem que é enviada após uma rota ser calculada. O nó que recebe esta mensagem atualiza sua cache e, sendo este o nó que iniciou a requisição da rota, encaminhamos a mensagem para o destino através desta rota. Caso este não tenha sido o nó que requisitou a rota, a mensagem é reencaminhada para o nó anterior a este na rota.

Na figura 4.12 tratamos a mensagem *comm*, que é enviada contendo uma rota (calculada ou obtida na cache do nó origem) para o nó destino. Se o nó que recebe esta mensagem não é o nó destino, verificamos se ainda existe a conexão entre este nó e o próximo na rota (*neighbours.contains(pacote#1.next)*). Existindo a conexão, a mensagem é normalmente reencaminhada para o próximo nó na rota. Caso a conexão tenha sido quebrada, uma mensagem de erro é en-

```
this?comm (pacote) ->
        if this != pacote#1.last then
                if neighbours.contains(pacote#1.next) then
                        pacote#1.next!comm(pacote);
                else
                        sender!packet_error(pacote);
                        pacote.remove(pacote.range(this,pacote#1.last));
                        neighbours!rr(pacote);
                endif
        endif
```

Figura 4.12: Tratamento da mensagem que envia pacote sobre rota já calculada

viada para o último nó que encaminhou esta mensagem (*sender!packet-error*). Além disso, removemos a parte invalidada da rota e ativamos novamente o subprotocolo de descobrimento de rotas para obter uma nova rota deste ponto em diante.

```
this?packet_error (pacote) ->
        if this == pacote#1.first then
                cache.remove(pacote);
        else
                cache.remove(cache.range(this,pacote#1.last));
                pacote#1.previous!packet_error(pacote);
        endif
```

Figura 4.13: Tratamento de mensagens de erro enviadas quando uma conexão é quebrada

PUC-Rio - Certificação Digital Nº 0115606/CA

Quando um nó recebe uma mensagem de erro (*packet-error*, na figura 4.13), este testa se foi ele quem iniciou a comunicação. Se sim, remove da cache esta rota armazenada, a qual passou a ser inválida. Caso não tenha sido este nó que iniciou a comunicação, este remove da cache o trecho invalidado e reencaminha a mensagem para o nó anterior na rota.

Como LEP ainda não tem comandos para restrição de ações no tempo (tempo-real), as situações onde o protocolo DSR recorre a *timeouts* não foram especificadas. Entretanto, soluções presentes na especificação que permitem a verificação do protocolo sem perda de generalidade. Por exemplo, o protocolo DSR prevê que rotas inseridas na cache tenham um tempo de validade. Em nossa especificação, uma rota é removida da cache sempre que um nó detecta ou recebe alguma mensagem informando uma conexão quebrada.

Verificação

Na verificação do protocolo DSR, especificamos propriedades que atestam o correto comportamento do protocolo. Por exemplo, podemos verificar se eventualmente alguma mensagem de erro é necessariamente enviada, possivelmente causada pela movimentação dos nós:

<> (p!packet-error)

, que é validada, já que movimentações podem ocorrer indiscriminadamente de acordo com o parâmetro *dynamic*. Podemos verificar também se sempre que o sub-protocolo que calcula rotas é ativado, então em algum momento o agente que pediu o cálculo consegue enviar a mensagem:

[] (p!rr(1) → <>p!comm(1))

, a qual também é validada.

Além destas, também podemos especificar propriedades que analisam a correta especificação do protocolo. Por exemplo, a propriedade à seguir verifica se sempre que um nó envia uma mensagem de requisição de rota *rr*, este verificou sua cache:

[] (p!rr → p$cache.contains$)

, que para esta especificação do DSR não é validada. Na figura 4.12 (recebimento da mensagem *comm*) podemos observar que a mensagem *rr* pode ser enviada e a cache não é verificada (*cache.contains*) em nenhum momento anterior. Este comportamento ocorre na figura 4.9, mas não ocorre na figura 4.10. O contra-exemplo retornado é:

$\text{some}_1\text{!rr(p)} \Rightarrow \text{p?rr} \Rightarrow \text{p!rr}$

Observe que, caso a propriedade valesse, o teste da cache deveria ocorrer entre os comandos *p?rr* e *p!rr*, o que para algum caso não foi verificado.

4.2 Comparação entre LEP e outras linguagens

Uma das razões iniciais que nos motivou a propormos uma linguagem com construções de alto-nível como pronomes foi o tamanho das especificações escritas manualmente em Promela. Como já argumentamos, especificações extensas dificultam o entendimento, a manutenção, o que compromete a confiança no que foi especificado e os resultados obtidos. De forma a ilustrar esta diferença, no Apêndice C listamos especificações do algoritmo de eleição de um líder num anel em diferentes linguagens. O gráfico abaixo (figura 4.14) nos dá uma idéia desta comparação. De forma a ter uma comparação justa, todas as especificações foram escritas por diferentes autores, os quais são citados no respectivo Apêndice.

Nesta comparação utilizamos a métrica NOS (*Number Of Statements* (FvS05)) baseada no número de comandos. Esta é uma variante da métrica

PUC-Rio - Certificação Digital Nº 0115606/CA

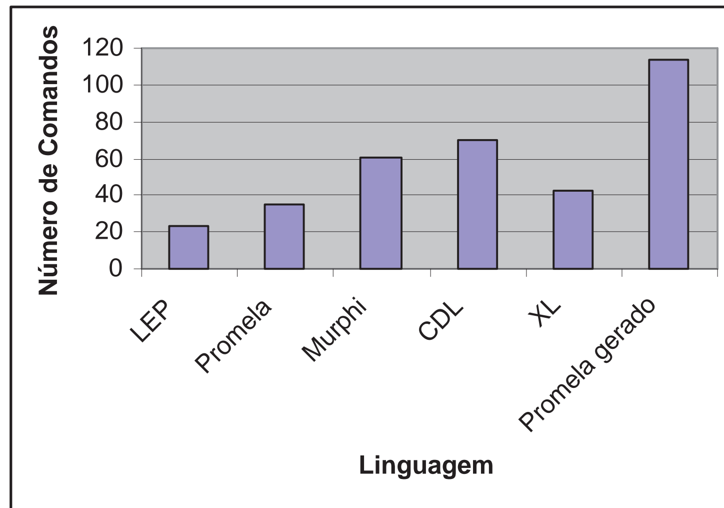

Figura 4.14: Número de Comandos nas especificações do algoritmo de Eleição de um Líder em diferentes linguagens

PUC-Rio - Certificação Digital Nº 0115606/CA

LOC (Kan03) que é baseada no número de linhas de código da especificação. Estas métricas, apesar de simples e inapropriadas para estimar a complexidade de um sistema a ser desenvolvido, podem ser utilizadas em nosso caso já que estamos analisando especificações já prontas. Nesta comparação foram considerados como comandos as atribuições, os condicionais (*if, else*), as transições (pré e pós-condições), as repetições (*do*) e os comandos de sincronismo de envio e recebimento de mensagens (por exemplo, *!, ?*) de Promela). Entretanto, não estão sendo incluídas especificações extras que são geradas como argumentos auxiliares de mensagens e declaração de variáveis ou estruturas de dados.

Na tabela 4.1 apresentamos uma relação entre cada pronome e o número aproximado de comandos do código equivalente gerados em Promela pela arquitetura. Nesta tabela nos baseamos para calcular o número de comandos existentes na especificação gerada em Promela dada no gráfico anterior (figura 4.14).

O fato de termos bem mais comandos na especificação gerada automaticamente em Promela do que na manual é justificado pela generalidade da versão automática. Isto quer dizer, se alterarmos a especificação em LEP para trabalharmos com uma topologia arbitrária, a mesma especificação em Promela seria gerada. Observe que na especificação em LEP (seção C.1), utilizamos o pronome *neighbours*, que ora se refere ao candidato seguinte (anel unidirecional), ora se refere a vizinhança de um candidato (rede arbitrária).

Na utilização de pronomes para expressar propriedades sobre o modelo, a simplificação não está refletida apenas no tamanho da fórmula, mas também no poder de expressão desta. Por exemplo, a fórmula abaixo:

Pronome em LEP	**Quantidade de Comandos em Promela**
this	1
sender	1
anyone	5
anyother	9
everyone	14*(NUM-TRANS + 1)
neighbours	10
none	1

Tabela 4.1: Tabela de Comparação entre LEP e Promela (NUM-TRANS contém o número de transições - *Pré-condição*→*Ação* - existentes no módulo onde o pronome *everyone* ocorre)

$$[]\ (\mathrm{p!acordado(everyone)} \rightarrow <>\mathrm{p?ack(any(k))})$$

define que sempre que um processo p envia uma mensagem *acordado* para todos os outros processos da rede, então eventualmente recebe pelo menos k mensagens do tipo *ack*. Colocando esta fórmula numa lógica temporal que não contenha pronomes, teríamos algo como:

$$[]\ (\mathrm{mh}_1\mathrm{!acordado} \wedge \mathrm{mh}_2\mathrm{!acordado} \wedge ... \wedge \mathrm{mh}_n\mathrm{!acordado} \rightarrow <>\mathrm{this?ack} \wedge ... \wedge \mathrm{this?ack})$$

, onde *this?ack* ocorre k vezes na fórmula e n é o tamanho da rede.

PUC-Rio - Certificação Digital Nº 0115606/CA