3
Implementação

Este capítulo apresenta a implementação do Ginga-NCL para dispositivos portáteis, baseada no perfil Basic DTV da linguagem NCL. Sua implementação foi baseada na implementação de referência do Ginga-NCL para dispositivos fixos, que é descrita resumidamente na Seção 3.1 desta dissertação. Da Seção 3.2 em diante, são analisados os problemas surgidos na nova implementação, as soluções oferecidas e as principais diferenças entre a implementação de referência para dispositivos fixos e a que está sendo apresentada nesta dissertação. A Seção 3.2 apresenta as API's básicas da plataforma Symbian C++, sua similaridade com a linguagem C++ e sua incompatibilidade com a biblioteca STL (Standard Template Library). A Seção 3.3 descreve os desafios superados na modificação do procedimento de parser do documento NCL para a versão portátil. A Seção 3.4 trata do uso e da necessidade da substituição da implementação da `thread` POSIX (Portable Operating System Interface for UNIX) por um outro recurso mais apropriado à plataforma Symbian C++, o Active Object, descrito na Seção 3.5. A Seção 3.6 explica como e por que o tratamento de âncoras temporais teve que ser alterado. A Seção 3.7 apresenta os exibidores de mídia com suporte no Symbian e o desafio de implementá-los, a fim de substituírem aqueles usados na implementação de referência para dispositivos fixos do Ginga-NCL. Por fim, a seção 3.8 tece algumas considerações finais acerca da implementação apresentada.

3.1.
A implementação de referência para terminais fixos

A implementação de referência do Ginga-NCL para terminais fixos foi desenvolvida em C++ para plataformas que utilizam o sistema operacional Linux. A Figura 1 e a Figura 2 ilustram a arquitetura do middleware Ginga.

Na Figura 1, a camada de serviço específico é apresentada. É possível observar a divisão do Ginga no ambiente declarativo Ginga-NCL, que corresponde à máquina de apresentação, e no ambiente procedural Ginga-J,

que corresponde à máquina de execução. Nesta dissertação, como já foi dito, apenas o Ginga-NCL é abordado, pois é o único ambiente obrigatório em receptores portáteis. Os módulos mais importantes da máquina de apresentação são o Conversor e o Gerenciador de Exibidores, que estão representados na Figura 1.

Figura 1: Camada de Serviço Específico do Ginga. Retirado adaptado de (Soares, Rodrigues e Moreno, 2007).

A Figura 2, por sua vez, apresenta a camada de núcleo comum da arquitetura Ginga. Essa camada possui módulos que servem tanto para o Ginga-NCL quanto para o Ginga-J, ou seja, ambas as máquinas fazem uso dessa camada. Os módulos mais importantes, representados na figura, são: Exibidores de Mídia, ou simplesmente Exibidores, Gerenciador do Módulo de Apresentação, Sintonizador, Filtro de Seções e Processador de Fluxos de Dados.

Figura 2: Núcleo Comum do Ginga. Retirado e adaptado de (Soares, Rodrigues e Moreno, 2007).

A máquina de apresentação é responsável por controlar a execução das aplicações NCL. A implementação dessa máquina, assim como a implementação do middleware como um todo, foi feita em C++ utilizando a STL a std lib e API's POSIX. Um documento NCL, descrevendo uma aplicação, deve ser processado pela máquina de apresentação antes que o controle da aplicação propriamente dita possa ser realizado.

Por ser uma aplicação XML, o processamento de um documento NCL se inicia pelo emprego de um parser XML. O módulo Conversor, que faz parte da máquina de apresentação, é responsável justamente por fazer esse processamento, traduzindo o documento NCL recebido da emissora, ou de qualquer outra fonte, em uma estrutura baseada no modelo NCM (Nested Context Model) (SOARES et all, 2003), o modelo conceitual da linguagem NCL. A partir dessa estrutura, a máquina de apresentação cria um modelo de execução que é, então, usado no controle da aplicação NCL. O módulo conversor foi implementado com base no framework para parsers DOM (Document Object Model) especificado em (Silva, Rodrigues e Soares, 2005) e fazendo uso da biblioteca libxerces-c, que implementa um parser DOM para documentos XML.

Uma vez tendo o modelo de execução montado, a máquina de apresentação inicia a apresentação da aplicação NCL. Para realizar a exibição das mídias da aplicação NCL em execução, a máquina de apresentação faz uso dos exibidores, ou objetos de execução. Os objetos de execução, pertencentes ao módulo Exibidores da Figura 2, são responsáveis por tratar as diferentes mídias que podem ser usadas em uma aplicação NCL. Os exibidores abstraem o uso das API's de mídia, que são específicas de plataforma, contribuindo para que o funcionamento da máquina de apresentação seja independente de plataforma. Existe um exibidor para cada tipo de mídia diferente e fica a cargo do Gerenciador de Exibidores, um outro módulo da máquina de apresentação, realizar a instanciação desses elementos. Uma vez instanciados, os conteúdos a serem exibidos devem ser apresentados em uma determinada região da tela do dispositivo em que a aplicação NCL está sendo executada, de acordo com a especificação da aplicação.

Cabe à máquina de apresentação determinar em qual região um exibidor associado a uma mídia qualquer deve ser instanciado. Em outras palavras, é de responsabilidade da máquina de apresentação o controle das regiões definidas em uma aplicação NCL e a associação dos exibidores a essas regiões.

Para fazer o acesso à tela do dispositivo, a máquina de apresentação faz uso do Gerenciador do Módulo de Apresentação. Tal gerenciador abstrai o acesso das possíveis camadas gráficas definidas pelo sistema de TV Digital e ainda o uso das API's de acesso à tela do dispositivo, que são específicas de plataforma. Essa abstração feita pelo Gerenciador do Módulo de Apresentação também contribui, assim, para que o funcionamento da máquina de apresentação seja independente de plataforma. O acesso às regiões da tela,

bem como sua manipulação, é feita pelo Gerenciador do Módulo de Apresentação através da biblioteca DirectFB (directfb.org | Main, 2008).

Diversas bibliotecas são utilizadas pelo módulo Exibidores. A decodificação por software de vídeos e de qualquer outro tipo de mídia contínua é feita com o uso da libxine. Imagens png, jpg, gif, tiff e bmp, são tratadas pelas bibliotecas libpng, libjpeg e libtiff. Por fim, documentos HTML são tratados pela telemidia-links. Mais informações sobre os exibidores e o acesso às camadas gráficas são encontradas em (Moreno, 2006: 74-81) e em (Moreno, 2006: 82-84).

Os módulos da camada Núcleo Comum usados na recepção de dados pelo fluxo de transporte (TS — transport stream), são o Sintonizador, o Filtro de Seções e o Processador de Fluxo de Dados.

O módulo Sintonizador permite que a máquina de apresentação sintonize em um canal específico com o objetivo de receber um fluxo de transporte enviado pela emissora. Informações mais detalhadas desse módulo podem ser encontradas em (Moreno, 2006: 66-69). O fluxo de transporte enviado pela emissora pode, por sua vez, transportar, ao mesmo tempo que o vídeo e o aúdio principal de um programa, dados diversos. Dados no formato de estrutura de sistema de arquivos são transportados pelo carrossel de objetos DSM-CC em seções privadas DSM-CC, conceito discutido com detalhes em (Moreno, 2006:22-32). O módulo Filtro de Seções é usado com o objetivo de filtrar as seções do fluxo de transporte, conforme detalhado em (Moreno, 2006: 69-70).

O módulo Processador de Fluxo de Dados, por sua vez, é usado no tratamento do protocolo DSM-CC que, entre outras funcionalidades, define o funcionamento do carrossel de objetos, usado na transmissão cíclica de dados (Moreno, 2006). O mesmo módulo é responsável por lidar também com eventos de fluxo, que também fazem parte do protocolo DSM-CC. Mais detalhes acerca desse protocolo e do Processador de Fluxo de Dados podem ser encontrados, respectivamente, em (Moreno, 2006: 32-39) e (Moreno, 2006: 70-74).

Dos elementos apresentados resumidamente nesta seção, a máquina de apresentação, o Gerenciador de Exibidores, o Conversor, o modulo Exibidores e o Gerenciador do Módulo de Apresentação foram implementados na versão do Ginga-NCL para dispositivos portáteis. As especificidades dessa implementação são apresentadas nas seções seguintes.

Como os receptores disponíveis no mercado não dispunham de recepção na modulação SBTVD-T (modulação ISDB), os módulos Sintonizador, Filtro de Seções e Processador de Fluxo de Dados — usados no tratamento do fluxo de transporte — não foram implementados, ficando como trabalhos futuros a serem

realizados. Para a simulação de funcionamento, programas de TV pré-armazenados no dispositivos foram utilizados.

3.2. Symbian C++, a API Base e a STL

O desenvolvimento de aplicações na plataforma do Symbian OS é feito com o uso da linguagem Symbian C++, que, por sua vez, é baseada na linguagem C++. Symbian C++ possui uma série de API's, mas a mais básica delas é a API Base (Symbian Ltd, Nokia, 2006), como indica o próprio nome. A linguagem Symbian C++ e a API Base possuem algumas diferenças da linguagem C++ e das API's básicas usadas na implementação de referência do Ginga-NCL para dispositivos fixos. Essas diferenças tiveram que ser tratadas com cuidado na implementação do Ginga-NCL para dispositivos portáteis. Esta seção apresenta, primeiramente, as principais diferenças da linguagem Symbian C++ e sua API Base para a linguagem C++ e suas API's básicas. Em seguida, apresenta como essas diferenças foram tratadas na atual implementação.

Um recurso comum muito utilizado na linguagem C++ são as exceções. O uso de exceções em Symbian C++ não era suportado inicialmente (Morley, 2007). Ao invés delas, o uso de recursos nativos da plataforma Symbian, os Leaves e as TRAPS (Symbian Ltd, Nokia, 2006), era a solução oferecida pela linguagem. Posteriormente, as exceções passaram a ser suportadas e até mesmo recomendadas por consumirem menos ciclos de CPU que os Leaves e TRAPS, e por serem compatíveis com o padrão da linguagem C++. Entretanto, existem algumas diferenças na implementação desse recurso no Symbian por conta do seu consumo de memória.

Cada exceção exige a alocação de um espaço de memória e, na linguagem C++, mais de uma exceção pode ser lançada ao mesmo tempo. Essa ação, chamada de lançamento aninhado de exceções, pode ocorrer quando uma exceção é lançada no destrutor de um objeto qualquer, como é explicado em (Morley, 2007). Como o lançamento de exceções em destrutores é algo plausível, torna-se impossível saber o quanto de memória uma aplicação precisará alocar para permitir o uso de exceções. Se, no entanto, o lançamento aninhado de exceções nunca ocorrer, ou seja, se exceções nunca forem lançadas em destrutores de objetos, é garantido que o espaço de memória necessário seja exatamente o de uma única exceção.

Como a alocação do espaço de memória poderia ser proibitivo se mais de uma exceção fosse lançada ao mesmo tempo, Symbian C++ proíbe o lançamento aninhado de exceções. Quando isso acontece, a aplicação que o fez é simplesmente abortada. Isso garante a necessidade de uma única alocação de memória para o uso de exceções ao longo de toda a vida do programa. Sendo assim, existe a necessidade de se remover quaisquer lançamentos aninhados de exceções de códigos portados para a plataforma Symbian. Essa resolução e mais detalhes sobre o uso de exceções em Symbian são encontrados em (Morley, 2007).

Felizmente, a implementação de referência do Ginga-NCL para dispositivos fixos faz pouco uso de exceções, e que nunca são lançadas de forma aninhada. Assim, o uso desse recurso não precisou ser modificado na implementação para dispositivos portáteis. Chama-se a atenção, no entanto, para o problema, quando do porte de outras implementações Ginga-NCL que possam vir a fazer uso de exceções aninhadas.

O tratamento de exceções foi a única diferença encontrada entre a linguagem Symbian C++ e a C++. Problemas maiores são vistos no uso da API Base do Symbian OS com o objetivo de se substituir as API's básicas usadas na implementação de referência do Ginga-NCL.

A API Base do Symbian possui as estruturas mais simples que podem ser usadas na implementação de uma aplicação qualquer. Nessa API, todos os tipos básicos encontrados na linguagem C++ são redefinidos. Muitos desses tipos são, na verdade, meras re-declarações. Como exemplo, tem-se o `TInt`, um tipo definido pela API Base, que é, na verdade, um `typedef` para o tipo básico `signed int`. A afirmação é verdadeira para todos os tipos básicos integrais e, portanto, nesses casos, é seguro fazer uso desses tipos diretamente. Nos demais casos, como o do `TChar`, substituto do `char`, são definidas novas classes na API Base, sendo o uso delas recomendado, mas não obrigatório, nem crítico. A recomendação de uso desses tipos específicos do Symbian serve mais para forçar os programadores a manterem uma padronização no código do que para aumentar a sua eficiência. Ou seja, essa recomendação não precisa ser seguida para fins de porte.

Com base nas afirmações do parágrafo anterior, todos os tipos básicos usados na implementação de referência do Ginga-NCL para dispositivos fixos, tanto na máquina de apresentação como em outros módulos, não foram mapeados em tipos Symbian C++, ou seja, o porte foi feito de forma direta. Por

outro lado, todo o código novo inserido na implementação fez uso exclusivo da API Base.

A implementação para dispositivos fixos do Ginga-NCL faz uso de `strings`, `threads` e `mutexes`, além de outras API's std lib e POSIX. Isso sim causou, de fato, um problema de porte, uma vez que a API Base tem a sua própria definição para esses elementos, que são incompatíveis com aqueles da std lib e do POSIX.

Para o problema mencionado no parágrafo anterior, foram criados o P.I.P.S. (P.I.P.S. Is Posix on Symbian) e o Open C. A Symbian Ltd, a pedido da comunidade de desenvolvimento Symbian, iniciou o desenvolvimento do P.I.P.S. com o objetivo de facilitar o porte de aplicações desenvolvidas em C/C++. Essa biblioteca, entretanto, não cobre todas as API's POSIX nem toda a std lib. Em decorrência, a Nokia desenvolveu o Open C, um superset do P.I.P.S. para o S60, cuja implementação cobre um pouco mais o oferecido pelas API's POSIX e a std lib, mas não ainda a sua totalidade. Maiores informações sobre o percentual das bibliotecas implementadas pelo P.I.P.S. e Open C podem ser encontradas em (Forum Nokia, 2007: 6).

Felizmente, na implementação do Ginga-NCL para dispositivos portáteis, o P.I.P.S. e o Open C cobriram o uso de todas as funcionalidades requeridas, o que inclui `strings`, `threads` e `mutexes`. A Seção 3.4 desta dissertação demonstra, entretanto, que o uso de `threads` não pôde ser mantido na implementação do Ginga-NCL para dispositivos portáteis.

A implementação de referência do Ginga-NCL para dispositivos fixos também faz muito uso de outros elementos complexos, como vetores, conjuntos e listas, lançando mão da API STL. O problema aqui é que a API Base define novas classes para esses elementos, que são completamente diferentes daquelas definidas na STL.

Uma solução para esse problema foi criar um mapeamento próprio das classes STL utilizadas na implementação de referência do Ginga-NCL para classes correspondentes da API Base. Como as classes da STL e da Base são muito diferentes entre si, tendo, normalmente, assinaturas de métodos muito distintas, o mapeamento proposto tornou-se muito difícil. No caso da classe STL `map`, não existe nem mesmo uma classe correspondente no Symbian que pudesse ser usada de forma adequada. A classe mais próxima da map definida pela API Base é a `TBTreeFix<Chave, Valor>` (Symbian Ltd, Nokia, 2006), que usa uma árvore B para armazenar Valores relacionados às suas respectivas Chaves. Contudo, essa API possui uma forte restrição, pois Valor e Chave

devem sempre possuir um tamanho fixo em bytes. Isso significa que todas as Chaves inseridas em um objeto `TBTreeFix` devem possuir um tamanho fixo igual a X, e que todos os Valores inseridos devem ter tamanho fixo igual a Y, onde X e Y são quaisquer números em bytes. Isso tornou a substituição da `map` pela `TBTreeFix` inoportuna, uma vez que os objetos usados pela implementação possuem tamanhos indefinidos. Seria necessário definir tamanhos máximos para todos os objetos e garantir que todos tivessem exatamente esses valores, mesmo que precisassem, na prática, ocupar espaços menores.

Uma opção muito mais simples foi implementar uma `map` própria, criando duas listas, uma para armazenar as Chaves e a outra para armazenar os Valores. As duas listas ficam relacionadas entre si e são ordenadas crescentemente por ordem de Chave. A solução proposta não causa gasto desnecessário de memória, fator importante no escopo dos dispositivos portáteis, e o tempo de busca por um elemento na lista não é substancial, uma ver que o uso das `maps` no Ginga-NCL é bem simples e com poucas entradas. A criação dessa `map` específica resolveu, portanto, o problema da incompatibilidade da classe `map` STL com a plataforma Symbian de forma eficiente.

Uma outra incompatibilidade encontrada foi no uso de `iterators`. Esse é um recurso provido pela STL usado para realizar interações em estruturas de lista. Esse recurso, entretanto, simplesmente não existe na API Base do Symbian. Diferentemente do caso das `maps`, a solução para esse problema foi substituir todas as ocorrências encontradas de uso de `iterators` por um outro procedimento de interação sobre listas equivalente.

Essa solução de mapeamento foi usada em uma parcela da implementação do Ginga-NCL para dispositivos portáteis com sucesso, mas demonstrou-se muito problemática. Se toda a implementação do Ginga-NCL fosse feita com o uso dessa solução de mapeamento, boa parcela do código original teria que ser desnecessariamente modificada. Isso poderia introduzir erros que, depois, seriam de difícil remoção.

Outros programadores da comunidade Symbian também experimentaram o mesmo problema e, tendo consciência disso, a Symbian Ltd começou a implementar um porte da STL para o sistema operacional Symbian. Todavia, essa implementação ainda não havia terminado quando o desenvolvimento do Ginga-NCL para dispositivos portáteis foi iniciado. Dessa forma, ou mantinha-se a solução do mapeamento das API's já descrita, o que consumiria muito tempo e poderia gerar erros, ou tentava-se realizar, por conta própria, o porte de alguma

implementação STL existente para o Symbian. Essa última opção poderia também consumir muito tempo. Além disso, a implementação STL escolhida para o porte teria que ter sido desenvolvida com a preocupação no uso de recursos, caso contrário, a implementação do Ginga-NCL poderia ficar muito pesada para ser executada nos dispositivos portáteis.

Um dos membros da comunidade Symbian, entretanto, fez um porte eficiente de uma implementação STL para o Symbian. Essa implementação é a STLPort (STLPort, 2001), que foi desenvolvida em ANSI C++ de maneira otimizada, visando ao máximo a eficiência. Com essas características, a API STL em questão pode ser portada para plataformas embarcadas ou com restrições de recursos sem muitos problemas. O porte dessa API para o Symbian pode ser obtido em (Jez, 2007). A sua instalação e uso são simples, bastando apenas fazer a referência correta aos `headers` e ligar estaticamente a única biblioteca necessária, a stlport_s.lib. Feito isso, o código com referências a STL compila e executa normalmente. Não foi observada nenhuma degradação do sistema devido ao uso dessa biblioteca. Sendo assim, a solução para o problema da STL foi considerado resolvido com o uso desse porte, tendo sido abandonada por completo a solução de mapeamento descrita.

3.3. Parser do Documento NCL

É possível enumerar dois tipos de parser XML: o SAX (Simple API for XML) e o DOM (Document Object Model) parser.

O primeiro oferece uma API simples e eficiente para o tratamento de documentos XML. O SAX percorre todo o documento apenas uma vez e, durante esse processo, lança eventos. Esses eventos indicam o início de uma tag, o fim dela, o início ou o fim de um documento como um todo, dentre outras ações. Para utilizar o SAX, uma aplicação precisa implementar funções que recebam e tratem esses eventos. Essas funções, quando chamadas, são supridas de todas as informações necessárias, como os atributos de uma tag. Por conta desse modelo, o SAX possui uma API leve, principalmente em termos de consumo de memória, pois, uma vez que o evento foi tratado, nada mais referente a ele fica alocado. O problema no uso dessa API é o fato do desenvolvimento tornar-se mais complicado. Não é possível, por exemplo, obter dados sobre uma tag que já foi tratada. Assim, se uma aplicação precisa de informações do XML em

diferentes momentos, ou se precisa adicionar alguma informação ao documento, o SAX pode não ser uma boa opção.

O DOM parser, por sua vez, monta um modelo de árvore em memória a partir de um documento XML qualquer. Esse modelo é como qualquer outro objeto, podendo ser alterado e usado sem grandes dificuldades. Por outro lado, como o DOM armazena e mantém o modelo em memória, usá-lo em dispositivos que possuem restrições de recursos não é muito indicado.

Estudos comparativos entre DOM e SAX são feitos em (Violleau, 2002), (Oren, 2002), (Devsphere, 2007) e (Franklin, 2007). Foram feitos testes empíricos de desempenho em relação ao consumo de memória e de processador nesses estudos, e os resultados demonstram a superioridade do SAX nesses aspectos. Entretanto, mesmo com as vantagens do SAX, a versão original do Conversor desenvolvida para a implementação de referência do Ginga-NCL em dispositivos fixos, descrito no Item 3.1, foi feita com o uso do DOM parser.

A natureza do Ginga-NCL aparentemente não exige que sejam feitos muitos acessos ao documento NCL, e qualquer alteração que precise ser feita na aplicação pode ser aplicada diretamente no seu modelo NCM associado. Essas características do Ginga-NCL fazem com que o uso do DOM parser não seja necessário. Em especial, levando-se em consideração que o estudo desta dissertação lida com dispositivos que possuem limitações de recursos, o uso do DOM parser torna-se oneroso. A Symbian e a própria comunidade de desenvolvimento recomendam sempre o uso do SAX para a realização do parser de documentos XML em dispositivos portáteis. Assim, dada a evidência da falta de necessidade do uso do DOM parser, do seu gasto elevado de recursos e das recomendações da Symbian, o seu uso no módulo Conversor foi substituído pelo parser SAX na implementação do Ginga-NCL para dispositivos portáteis.

Sabe-se que o módulo Conversor da implementação de referência realizava o parser do documento NCL fazendo uso de um framework para parsers DOM especificado em (Silva, Rodrigues e Soares). Esse framework teve que ser estudado para que o funcionamento do módulo em questão pudesse ser reformulado a fim de se remover as particularidades existentes do parser DOM. Por outro lado, a parte específica do Ginga-NCL, que cria a estrutura NCM, foi mantida e adaptada para funcionar com o parser SAX. Essa adaptação gerou, para cada tag presente na linguagem NCL, uma função específica que cria o elemento NCM correspondente.

No Symbian, a API que realiza o parser de documentos XML não é genérica. No caso da plataforma S60, essa API se resume à classe `CParser`, um parser SAX de documentos XML. Para ser iniciado, o parser SAX em questão recebe o documento XML, no caso um documento NCL, e o analisa do início até o fim em busca de tags. Quando o parser descobre a existência de uma nova tag, uma notificação é feita através de uma chamada a função `OnStartElementL`, que, por sua vez, chama a função específica da tag encontrada, que irá criar o elemento da estrutura NCM correspondente. O primeiro problema encontrado foi o fato das chamadas feitas à função `OnStartElementL` serem independentes, o que impossibilita saber qual tag foi tratada antes ou qual será tratada depois. Para exemplificar o problema, a Figura 3 mostra um documento NCL simples.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <ncl id="Apresentacao01"
3  xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
4
5     <head>
6
7
8        <regionBase>
9           <region id="rgVideo" left="0%" top="0%"
10                 width="100%" height="100%"/>
11       </regionBase>
12
13       <descriptorBase>
14          <descriptor id="dVideo" region="rgVideo" />
15       </descriptorBase>
16
17    </head>
18
19
20    <body>
21
22          <port id="pInicio1" component="video"/>
23
24          <media type="video/3gp" id="video"
25                 src="whatisthematrix.3gp" descriptor="dVideo"/>
26
27    </body>
28 </ncl>
```

Figura 3: Exemplo simples de uma aplicação NCL.

Neste documento, um elemento <media> encontra-se dentro de um elemento <body>, que, por sua vez, está contido em um documento NCL. Na estrutura NCM, isso é representado por um objeto chamado `NclDocument`, que contém um `Context`, representando o nó body, e que, por sua vez, deve conter o nó de media com id igual a video. O parser precisa, portanto, criar essa estrutura e, para isso, precisa saber quem contém quem. Quando o parser SAX

chama a função `OnStartElementL` para o nó media, não existe qualquer forma de saber se esse nó faz parte do body ou de um outro nó de contexto. O parser perde essa informação e, portanto, seria impossível criar a estrutura NCM requerida. Esse problema não acontece no uso do parser DOM, uma vez que é montada uma árvore representando todo o documento XML. Nesse caso, basta fazer uma busca em profundidade nessa árvore para criar toda a estrutura do modelo NCM.

No caso do parser SAX, a solução para o problema foi pensar no procedimento de parser, que percorre o arquivo do início até o seu fim, como uma busca em profundidade propriamente dita. A Figura 4, baseada na NCL da Figura 3, ajuda a ilustrar esse pensamento onde a estrutura do documento NCL é vista como uma árvore.

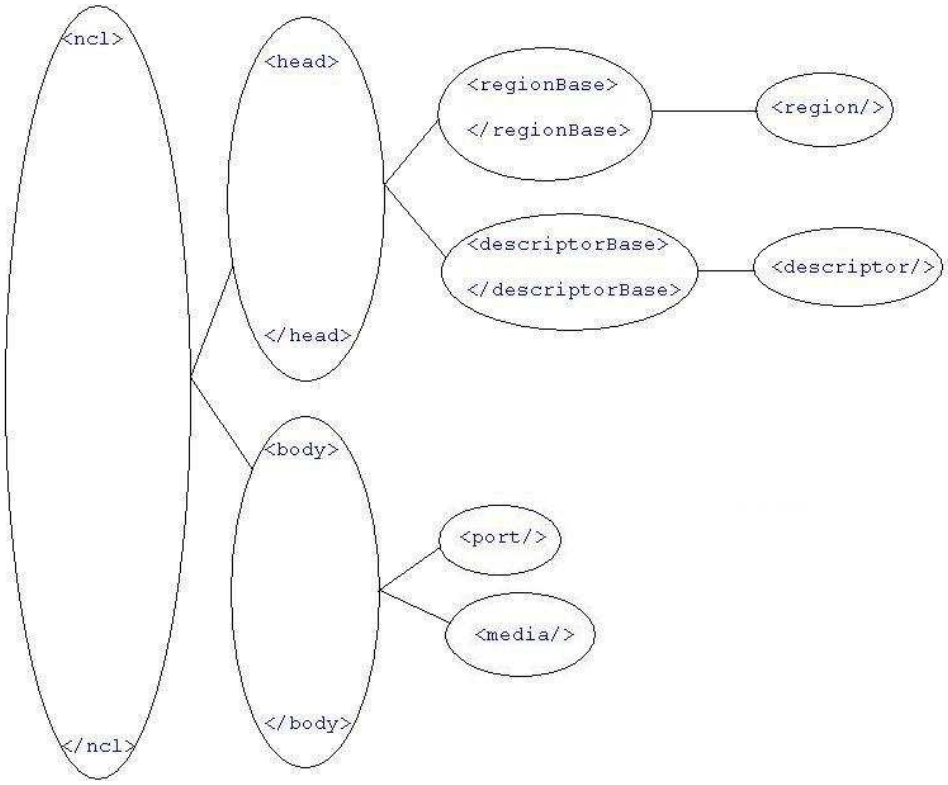

Figura 4: Documento NCL como uma árvore.

Para tratar a estrutura do documento NCL como uma árvore, duas informações são importantes: o início e o fim de uma tag. Pode-se dizer como as tags estão compostas (aninhadas) no documento NCL usando apenas essas duas informações. Por exemplo, olhando a Figura 4, sabe-se que “regionBase” e “descriptorBase” fazem parte do “head”, porque essa tag começa antes e termina depois das outras duas. Em suma, toda tag pai da composição inicia antes e termina depois das suas tags filhas. Assim, bastariam essas duas

informações para que se fosse possível enxergar o documento como uma árvore. As funções `OnStartElementL` e `OnEndElementL` do parser SAX em questão podem ser usadas exatamente para alcançar esse objetivo, pois a primeira informa o início, e a segunda, o fim de uma tag. Levando-se em conta o que já foi dito, é possível perceber que o procedimento de parser, que lê o arquivo NCL de cima para baixo, age, portanto, da mesma forma que uma busca em profundidade.

Como já foi dito, para montar a estrutura NCM corretamente, é preciso saber quem contém quem, e o primeiro passo para se descobrir isso é tratar o parser SAX como uma busca em profundidade, como apresentado.

É possível usar uma estrutura de pilha para o armazenamento do caminhamento feito por uma busca em profundidade. Esse procedimento, em si, é bem conhecido e, se aplicado, quando a busca exemplificada alcançar o elemento desejado, teremos na pilha dos elementos todos os seus ancestrais. Essa é exatamente a informação necessária para a construção da estrutura NCM. Basta, portanto, acessar a pilha para descobrir a qual nó o elemento corrente pertence.

Quando uma tag é tratada, o objeto NCM correspondente deve ser inserido na pilha. Quando a tag termina, culminando na chamada da função `OnEndElementL`, o elemento do topo da pilha deve ser removido. É garantido que o topo da pilha sempre corresponde ao elemento que está terminando, pois toda tag que é aberta deve ser fechada, caso contrário, o documento NCL estará inválido (mal formado). Ressalta-se que, para passar pelo procedimento de parser aqui descrito, o documento NCL precisa ter sido validado antes.

Considerando a NCL presente na Figura 3 sem a tag “head” e suas filhas, o procedimento parser descrito funcionaria da seguinte forma:

1- Início do Parser.
2- `OnStartElementL` da tag “ncl”. O elemento NCM `NclDocument` é armazenado na pilha.
3- `OnStartElementL` da tag “body”. O elemento NCM `Context` é inserido dentro do elemento que se encontra no topo da pilha, que, no caso, é o elemento `NclDocument`. Depois disso, `Context` é inserido na pilha.
4- `OnStartElementL` da tag “port”. O elemento NCM `Port` é inserido dentro do elemento que se encontra no topo da pilha, que, no caso, é o elemento `Context`. Depois disso, `Port` é inserido na pilha.
5- `OnEndElementL` da tag “port”. O topo da pilha, que contém o elemento Port, é removido.

6- `OnStartElementL` da tag “media”. O elemento NCM `ContentNode` é inserido dentro do elemento que se encontra no topo da pilha, que no caso é o elemento `Context`. Depois disso, `ContentNode` é inserido na pilha.
7- `OnEndElementL` da tag “media”. O topo da pilha, que contém o elemento `ContentNode`, é removido.
8- `OnEndElementL` da tag “body”. O topo da pilha, que contém o elemento Context, é removido.
9- `OnEndElementL` da tag “ncl”. O topo da pilha, que contém o elemento `NclDocument`, é removido.
10- Fim do parser.

O procedimento descrito estaria correto se não fosse por um segundo problema. A tag “media” precisa ser tratada antes da “port”, por ser referenciada por ela. Existem, na NCL, algumas tags semelhantes a essa que dependem de outras. Sendo a NCL uma linguagem declarativa baseada em XML, a ordem em que as tags aparecem no documento, exceto a “head” e a “body”, não deve influenciar no comportamento da aplicação resultante.

Para contornar esse problema duas soluções foram pensadas. A primeira seria realizar mais de uma vez o parser no documento NCL. O número de vezes seria determinado pelo nível de dependência das tags. Por exemplo, uma tag “port” depende da “media”, porque faz referência a um elemento desse tipo. Nesse caso, seriam necessárias duas passadas completas pelo documento NCL, uma para identificar as tags “media” e outra para tratar a tag “port”. A fim de evitar uso desnecessário de processamento, essa opção foi descartada. A segunda solução seria armazenar, temporariamente, as tags dependentes encontradas, que seriam tratadas depois, em um momento mais oportuno. Essa opção foi implementada devido ao fato da estrutura de armazenamento ser pequena e do seu descarte ser completamente efetuado após o tratamento das tags dependentes.

Para entender o funcionamento da solução proposta, é preciso saber como funcionam as relações de dependência entre tags NCL. As relações de dependência são sempre entre duas irmãs de uma mesma relação de composição, onde uma delas é sempre independente. Existe uma exceção dessa regra para as tags “medias” que usam o parâmetro “refer”. Nesse caso, a relação de dependência pode ocorrer entre tags de composições diferentes. A implementação do Ginga-NCL para dispositivos fixos já possui um tratamento

especial específico para esse caso que evita o acontecimento de erros e que foi, portanto, mantido.

Uma outra exceção é a dependência que tags contidas no elemento <body> possuem de outras pertencentes ao elemento <head>. Isso não gera problemas, pois é garantido que todas as tags contidas no head serão tratadas antes daquelas pertencentes ao body. Isso é verdade porque o nó head deve obrigatoriamente ser sempre declarado antes do body, e o procedimento de parser é sempre feito do início até o fim do documento ncl.

Para o caso geral de dependência, onde tags do nó body dependem de outras tags do nó body ou onde tags do nó head dependem de outras do nó head, é preciso esperar que o objeto ao qual a tag dependente faz referência seja tratado para, somente então, lidar com ela. Considerando, portanto, o caso geral, é garantido que, ao término do tratamento da tag pai de uma composição, todas as suas tags filhas independentes tenham sido criadas. Restariam as suas filhas dependentes, mas, como as suas irmãs já foram tratadas, é seguro, nesse momento, cuidar delas. Sendo assim, tags dependentes são sempre tratadas no momento em que a sua tag pai na relação de composição esteja terminando. Para isso, é usada a estrutura de armazenamento representada na Figura 5.

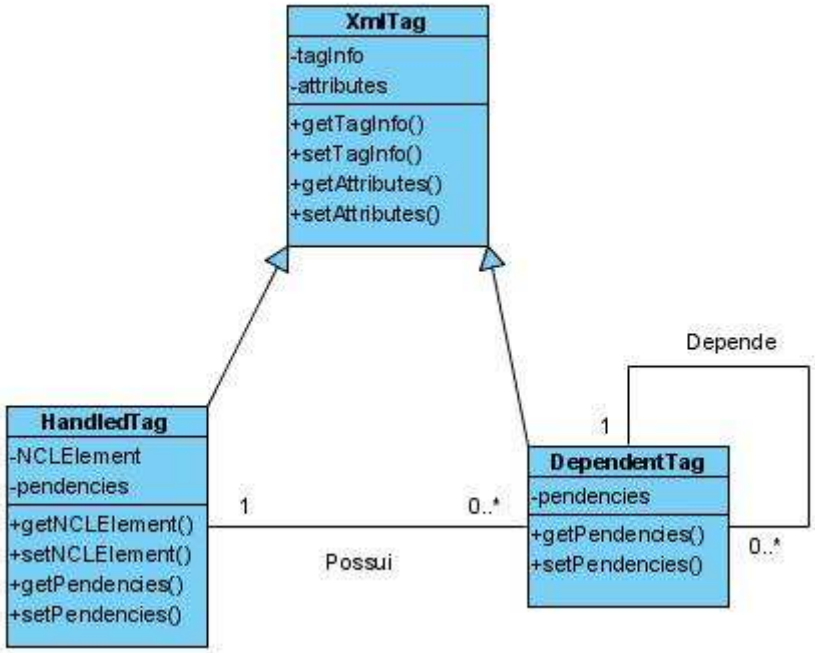

Figura 5: Diagrama de Classes da estrutura usada para resolver o problema das dependências entre tags.

A classe `HandledTag` armazena o objeto NCM referente à uma tag que já tenha sido tratada na sua propriedade de nome `NCLElement` e é, então, inserida

na pilha. Quando uma tag filha dependente é encontrada, uma instância da classe `DependentTag` é criada para armazená-la. Esse objeto é, por sua vez, inserido na lista de pendências da `HandledTag` que armazena o objeto NCM da sua tag pai na composição. Uma `DependentTag` também possui uma lista de pendências, pois, enquanto a tag dependente não tiver sido tratada, todas as suas tags filhas também não poderão ser e, portanto, deverão ser colocadas na sua lista de pendências como tags dependentes. Quando uma tag pai independente termina, a sua lista de pendências é tratada. O mesmo é feito com a lista de pendências de cada tag dependente que for tratada. Para exemplificar o procedimento de parser, agora sem erros, considere novamente a NCL presente na Figura 3. O parser, apenas para a tag “body”, funcionaria da seguinte forma:

1- Início do Parser.
2- `OnStartElementL` da tag “ncl”. O elemento NCM `NclDocument` é armazenado na pilha.
3- `OnStartElementL` da tag “body”. O elemento NCM `Context` é inserido dentro do elemento que se encontra no topo da pilha, que, no caso, é o elemento `NclDocument`. Depois disso, `Context` é inserido na pilha.
4- `OnStartElementL` da tag “port”. O elemento NCM `Port` é uma tag dependente e é inserido na lista de pendências do objeto `Context`, que representa a tag “body”. Depois disso, `Port` é inserido na pilha.
5- `OnEndElementL` da tag “port”. O topo da pilha, que contém o elemento `Port`, é removido. Como esta é uma tag dependente, a lista de pendências não é tratada agora.
6- `OnStartElementL` da tag “media”. O elemento NCM `ContentNode` é inserido dentro do elemento que se encontra no topo da pilha, que no caso é o elemento `Context`. Depois disso, `ContentNode` é inserido na pilha.
7- `OnEndElementL` da tag “media”. O topo da pilha, que contém o elemento `ContentNode`, é removido. Sua lista de pendências está vazia, portanto nada é feito.
8- `OnEndElementL` da tag “body”. O topo da pilha, que contém o elemento `Context`, é removido e a sua lista de pendências é resolvida. O único elemento existente é o “port”, que pode ser tratado agora, uma vez que “media” já foi resolvida. A lista de

pendências do “port” também deve ser resolvida nesse momento, mas, como ela está vazia, nada é feito.

9- `OnEndElementL` da tag “ncl”. O topo da pilha, que contém o elemento `NclDocument`, é removido. Sua lista de pendências está vazia, portanto nada é feito.

10- Fim do parser.

3.4. O Uso de Threads

“Uma thread pode ser definida como uma sub-rotina de um programa que pode ser executada de forma assíncrona, ou seja, executada paralelamente ao programa chamador”. (Berenger, Maia, 2002: 86) Em um ambiente multithread, um processo associado a uma aplicação possui pelo menos uma `thread`, chamada de `thread` principal. Uma aplicação pode ser totalmente executada na `thread` principal, ou ter o seu processamento dividido em duas ou mais `threads` diferentes. As diferentes `threads` de uma aplicação podem ser executadas concorrentemente ou, no caso de existirem múltiplos processadores, paralelamente.

“O conceito de concorrência é o princípio básico para o projeto e a implementação dos sistemas multiprogramáveis” (Berenger, Maia, 2002: 40) ou multitarefa, onde os programas compartilham os recursos computacionais disponíveis. O Symbian é um sistema operacional multitarefa que implementa as suas `threads` em modo kernel (Harrison, Richard et al, 2004). Além das `threads`, o Symbian também oferece os Active Objects, que podem ser usados com o mesmo objetivo das `threads`. A diferença entre os dois recursos é que os Actives Objects são como `threads` implementadas em modo usuário e que não sofrem qualquer tipo de preempção (Harrison, Richard et al, 2004).

A implementação do Ginga-NCL para terminais fixos faz muito uso de `threads` por meio das API’s POSIX. Essas `threads` são usadas tanto para permitir ações que executam em paralelo, importantes em aplicações que exijam sincronismo, quanto para resolver outros problemas, como o do tratamento de âncoras temporais, detalhado na Seção 3.6. As API’s de criação e manipulação das `threads` no Symbian, entretanto, são diferentes daquelas oferecidas pelo POSIX. Como visto na Seção 3.2, o uso de `threads` POSIX no Symbian só foi possível graças ao P.I.P.S e ao Open C. Assim, o porte das classes que se utilizam de `threads` POSIX pôde ser feito de forma direta, ou seja, nada da

implementação de referência do Ginga-NCL para dispositivos fixos teve que ser mudado.

O uso de `threads`, entretanto, consome muitos recursos, principalmente os de memória, pois, para cada nova `thread`, uma nova pilha de execução é criada. Por isso, em um ambiente de dispositivos com limitações de recursos, o uso de `threads` normalmente não é recomendado. No sistema operacional Symbian, em específico, a forma preferencial de se implementar uma aplicação multitarefa, como o Ginga-NCL, não é com o uso das `threads`, mas sim com o uso dos Actives Objects (Harrison, Richard et al, 2004: 41). Entretanto, a fim de se facilitar o procedimento de porte, tentou-se manter o uso das `threads`. Assim seria possível acelerar o processo de desenvolvimento e garantir a correção do programa, uma vez que as características da implementação anterior do Ginga-NCL, amplamente testado, seriam mantidas.

Um primeiro obstáculo encontrado para manter o uso das `threads` é o fato de que muitos dos recursos oferecidos pelo Symbian só podem ser usados na `thread` principal do processo de uma aplicação. Esse problema tem relação com o modelo Client-Server Framework (Stichbury, 2004: 167) amplamente utilizado no Symbian para a obtenção, manipulação e posterior liberação de recursos do sistema. Muitos dos recursos só podem ser acessados através desse modelo, onde um cliente solicita a um servidor um objeto que poderá ser usado na manipulação de um recurso qualquer. Esse objeto só pode ser usado na `thread` onde foi requisitado, e é somente através dele que o recurso associado pode ser acessado. Vídeo, áudio e imagem são alguns exemplos de recursos geridos pelo Client-Server Framework. Uma lista mais detalhada de recursos geridos por esse modelo é encontrada em (Harrison, Richard et al, 2003: 501).

Ao iniciar uma aplicação Symbian, todo um ambiente de execução predefinido é carregado (Harrison, Richard et al, 2004: 99). Esse ambiente inclui, dentre outras coisas, controles para entrada de dados pelo usuário, gráficos e acesso ao sistema de arquivos. Assim, recursos como os de acesso à tela do dispositivo e exibição de gráficos em geral já são carregados por padrão na `thread` principal do processo de qualquer aplicação Symbian (Harrison, Richard et al, 2004). Como estes são recursos que são usados seguindo o modelo Client-Server Framework, outras `threads` do processo não podem manipulá-los. Assim, a instanciação e manipulação de objetos específicos de exibição do Ginga-NCL, como o exibidor de vídeo, precisam ser feitas na `thread` principal do processo de uma aplicação Symbian. Como a implementação de referência do Ginga-NCL para dispositivos fixos lança diversas `threads` ao longo da sua

execução, e muitas delas são responsáveis exatamente por instanciar e manipular objetos de exibição, introduziu-se um problema que teve de ser contornado.

Para solucionar o problema, e para manter o uso das `threads`, foi criado um esquema de requisição entre as `threads` genéricas do processo e a `thread` principal, onde as primeiras requisitariam instanciações ou ações pré-definidas de objetos, e a segunda as atenderia. A idéia consiste em deixar a `thread` principal em espera pelas requisições, enquanto as outras `threads` são executadas e realizam essas requisições durante o seu processamento.

A implementação desse mecanismo faz uso de um vetor de requisições compartilhado entre todas as `threads`. Uma ou mais `threads` podem, então, armazenar uma ou mais requisições nesse vetor, sinalizando apropriadamente a sua ação. Ao ter conhecimento dessa ação, a `thread` principal trata todas as requisições pendentes na lista em questão. Ao final, a `thread` principal volta a aguardar por novas requisições.

As `threads` podem fazer requisições a funções sem retorno, como a de uma ação de “play” de vídeo, e depois continuar a sua execução normalmente. Se for necessária uma requisição a funções com retorno de valor, como métodos do tipo `get`, ou a instanciação de objetos, a `thread` deve, obrigatoriamente, esperar pelo término da requisição, a fim de obter o retorno da função ou o objeto instanciado. Nesses casos, a `thread` principal armazena, em uma variável compartilhada, o retorno da função ou o objeto instanciado tão logo a requisição for finalizada. Através dessa variável compartilhada, a `thread` requisitante é capaz de recuperar a informação requerida.

A solução, embora tenha resolvido o problema, não foi suficiente para manter o uso das `threads` na implementação do Ginga-NCL para dispositivos portáteis. Na prática, a implementação das `threads` fazendo uso da solução descrita mostrou-se extremamente lenta. Isso aconteceu a tal ponto de não ser possível manter o sincronismo em aplicações muito simples, como, por exemplo, a exibição de três imagens na tela. Infelizmente, a compatibilidade com a versão anterior do Ginga-NCL no uso de `threads` não pôde ser mantida na implementação proposta. Em seu lugar, os Active Objects foram usados, conforme descrito na seção seguinte.

3.5. Active Objects

Como foi abordado na seção anterior, o uso de threads não pode ser mantido pela implementação proposta. Ao invés delas, optou-se por fazer uso dos Actives Objects (Stichbury, 2004). Fazendo uma comparação entre os dois recursos, é possível perceber a vantagem de se usar os Actives Objects. A transferência do controle de um Active Object para outro, por exemplo, pode ser até dez vezes mais rápida do que no caso de uma `thread` (Stichbury, 2004: 113). O consumo de memória é outra vantagem: enquanto uma `thread` pode consumir, aproximadamente, 4Kb de memória de kernel e 8 Kb de pilha de execução; um Active Object consome, em média, algumas centenas de Bytes ou até menos (Stichbury, 2004: 113). A última vantagem que pode ser citada é que não existe a preocupação com a proteção de dados compartilhados, uma vez que os Active Objects se encontram em uma mesma `thread` (Harrison, Richard et al, 2004). O único caso onde o uso dos Active Objects não é apropriado é quando se deseja criar aplicações de tempo real que exigem tempos de resposta muito pequenos. Nesses casos, o melhor é implementar a aplicação de tempo real em uma `thread` de alta prioridade (Stichbury, 2004).

Em Symbian C++, entretanto, os Active Objects são usados normalmente com o objetivo de controlar as chamadas às funções assíncronas e o seu término. Portanto, para entender melhor o funcionamento dos Active Objects é interessante entender, antes, o que são funções assíncronas. Uma função assíncrona é aquela que retorna imediatamente após a sua chamada e executa em paralelo ao procedimento que a chamou, normalmente em uma `thread` separada. Esse procedimento chamador pode bloquear o seu processamento ou continuar e realizar outras ações enquanto aguarda pelo fim da função assíncrona. Quando a função assíncrona termina, um evento é lançado indicando o seu sucesso ou erro. Esse evento deve ser capturado e tratado pelo procedimento que chamou a função assíncrona.

Um Active Object é um objeto que estende a classe `CActive` (Symbian Ltd, Nokia, 2006). As funções assíncronas controladas por um Active Object são nele encapsuladas, e só devem ser chamadas através dele. Embora um Active Object possa encapsular mais de uma função assíncrona, apenas uma pode ser executada por vez. Ou seja, enquanto uma função assíncrona de um Active Object estiver sendo processada, uma outra do mesmo objeto não pode ser chamada. Se essa restrição for desrespeitada, existem três comportamentos

possíveis: a aplicação é finalizada, a nova chamada não é realizada ou o processamento da última função assíncrona é finalizado dando lugar à nova.

No Active Object também é implementado o método `RunL`, que é responsável por tratar o término das suas funções assíncronas. Esse método pode ser usado para analisar o resultado da chamada à função assíncrona, realizar algumas finalizações ou até mesmo fazer outra chamada. Em suma, pode-se fazer qualquer ação que se faça necessária.

Um Active Object é, portanto, de forma simplificada, um objeto que possui um ou mais métodos que implementam ou realizam chamadas a quaisquer funções assíncronas, e um outro método responsável por tratar das suas finalizações. Uma API de vídeo, por exemplo, pode ser um Active Object, sendo a sua função de Open assíncrona. Ao ser chamada pela aplicação, a função em questão tem iniciada a sua execução em paralelo. A aplicação continua o seu processamento e, em um determinado momento, a função Open termina a sua execução. Então, o método `RunL` da API de vídeo em questão é chamado, e faz as finalizações necessárias. Ele pode, por exemplo, emitir uma mensagem de erro ao usuário caso o processamento da função Open tenha falhado.

Um outro objeto, o Active Scheduler, tem como responsabilidade chamar o método `RunL` do Active Object quando uma das funções assíncronas desse elemento tiver terminado. Quando isso acontece, diz-se que o Active Scheduler fez o escalonamento de um Active Object. Existe apenas uma instância do Active Scheduler por `thread` e o seu funcionamento é muito parecido com o de um escalonador comum, tendo como diferença básica o fato dele não realizar qualquer tipo de preempção.

Para fazer o procedimento descrito, o Active Scheduler define um semáforo global à `thread` em que ele se encontra. Esse semáforo deve ser sinalizado pelas funções assíncronas quando o processamento delas tiver terminado. Dessa forma, o Active Scheduler sabe quando uma função assíncrona terminou e que, portanto, um Active Object deve ser escalonado através da chamada ao seu método `RunL`. A função assíncrona também precisa alterar o valor de uma variável interna do seu Active Object associado, denominada `iStatus`. Essa variável indica se o Active Object tem uma função assíncrona finalizada ou ainda em execução. Essa ação é necessária porque é possível existir mais de um Active Object gerido por um Active Scheduler, e a simples sinalização do semáforo em si não indica qual Active Object teve a sua função assíncrona finalizada. Ao alterar o valor da variável em questão,

indicando que o Active Object possui uma função assíncrona já finalizada, o Active Scheduler é capaz de descobrir qual Active Object pode ser escalonado.

O Active Scheduler fica, então, aguardando até que uma ou mais funções assíncronas terminem e façam a sinalização. É importante ressaltar que, existindo mais de um Active Object, mais de uma função assíncrona pode estar sendo processada ao mesmo tempo. Em um cenário desses, duas ou mais dessas funções podem ter terminado antes que o Active Scheduler tenha tido a oportunidade de tratar a primeira delas. Por conta disso, quando o Active Scheduler for tratar o término de uma função assíncrona, é normal que exista mais de um Active Object elegível a ter a sua função `RunL` executada. É interessante, portanto, que o Active Scheduler faça o escalonamento daquele Active Object que for mais crítico para o sistema, ou seja, aquele que for mais prioritário. Dessa forma, todo Active Object precisa determinar, no momento da sua construção, uma prioridade para si mesmo de tal forma que o Active Scheduler possa usar essa informação no momento em que estiver fazendo o escalonamento. Um outro ponto importante é que os Active Objects devem ser registrados junto ao Active Scheduler para que possam ser escalonados. Normalmente, um Active Object faz esse registro logo na sua construção, mas isso pode ser feito em qualquer momento, desde que antes da chamada a uma função assíncrona. O Active Scheduler, por sua vez, mantém uma lista de Active Objects registrados com base nas suas prioridades, e faz o escalonamento com base nessa lista.

Uma vez tendo recebido a sinalização do término de uma função assíncrona, o Active Scheduler escalona o Active Object mais prioritário que possui uma função desse tipo finalizada e faz a chamada ao seu método `RunL`. Esse método será executado sem interrupções, até o seu fim, uma vez que não existe preempção no uso dos Actives Objects. Quando o método `RunL` termina, o Active Scheduler volta a aguardar por novas sinalizações. O funcionamento do Active Scheduler pode ser, portanto, resumido em um loop pelo qual ele:

1- Aguarda por uma sinalização de uma função assíncrona que tenha terminado;
2- Recebe a sinalização e busca pelo Active Object de maior prioridade que possui uma função assíncrona finalizada, o que é possível de se descobrir ao se consultar o valor da variável `iStatus` dos Actives Objects;
3- Ao final da busca, chama o método `RunL` do objeto escalonado;

4- Quando o método `RunL` terminar, volta a aguardar por uma nova sinalização.

A qualquer momento nesse loop, novos Active Objects podem se registrar junto ao Active Scheduler. Informações mais detalhadas sobre os Active Objects podem ser encontradas em (Stichbury, 2004: 127).

Na implementação apresentada, entretanto, o mais interessante é saber como fazer uso do Active Object com o objetivo de suprimir o uso das `threads` POSIX tão utilizadas na implementação de referência do middleware Ginga-NCL para dispositivos fixos. Para tanto, é recomendado fazer uso dos Background Active Objects (Symbian Ltd, Nokia, 2006), que são Active Objects de baixa prioridade.

A idéia é encarar o Background Active Object como uma `thread` em si. No seu método `RunL` deve ser implementado o procedimento que seria implementado na função `run` da `thread` POSIX. Como não existe preempção no uso dos Active Objects, o método `RunL` não pode gastar muito tempo durante o seu processamento, pois isso impediria que outros Active Objects, alguns talvez mais prioritários, fossem escalonados por um longo período de tempo. O procedimento deve, então, ser implementado no método `RunL` do Background Active Object de modo que não consuma muito tempo de processamento. Se o procedimento for muito pesado, ele deve ser dividido em N partes diferentes. Cada uma dessas N partes deve ser leve o suficiente para não consumir muito tempo de processamento do método `RunL`. A idéia é que a execução dessas N partes seja feita de forma intercalada, com o objetivo de dar ao Active Scheduler a oportunidade de escalonar um outro Active Object de maior prioridade. Por isso que é importante que o Background Active Object possua uma prioridade baixa, assim, entre a execução de uma das N partes do procedimento e outra, é garantido que outros Actives Objects mais prioritários serão escalonados. Caso o procedimento seja leve o suficiente para ser executado todo de uma vez, esses cuidados não precisam ser tomados.

Como já mencionado, um Active Object só fica elegível pelo Active Scheduler quando o semáforo desse elemento é sinalizado por uma função assíncrona e quando o valor da sua variável `iStatus` é modificado de forma apropriada. Dessa forma, para iniciar a execução do procedimento implementado no método `RunL` de um Background Active Object, um outro método, que aqui será chamado de `start`, deve ser implementado em substituição a função assíncrona. A chamada ao método `start` deve fazer com que o Background Active Object em questão fique elegível pelo Active

Scheduler. Para isso, o método em questão precisa apenas sinalizar o semáforo controlado pelo Active Scheduler e atualizar a variável `iStatus` como já descrito nesta seção. Ao chamar esse método, o Background Active Object fica elegível pelo Active Scheduler e o seu método `RunL` será chamado assim que nenhum Active Object mais prioritário estiver na fila.

Quando o método `RunL` é chamado, o procedimento implementado nele começa a ser executado. Se houve a necessidade de dividi-lo em N partes, uma dessas partes é executada e, então, o método `start` deve ser novamente chamado, fazendo o Background Active Object ficar mais uma vez elegível pelo Active Scheduler. Isso permite que o método `RunL` seja chamado novamente para que uma outra das N partes do procedimento em questão seja executada. Esse procedimento todo se repete até que todo o método `RunL` tenha sido executado e o procedimento implementado nele tenha, conseqüentemente, terminado. Se não houve a necessidade de dividir o procedimento em mais de uma parte, então ele é completamente executado logo na primeira chamada ao método `RunL`.

O único problema da técnica apresentada é modelar, quando necessário, o procedimento a ser implementado no método `RunL` adequadamente de tal forma que se possa dividi-lo em N partes.

Na implementação Ginga-NCL para dispositivos fixos, o uso de `threads` POSIX possui uma abstração que é representada por uma classe chamada `Thread`. Essa classe é usada pela máquina de apresentação quando é necessário executar ações concorrentemente. Um exemplo é quando se deseja implementar um conector onde estejam definidas ações que devem ser executadas em paralelo (ABNT/CEET-00:001.85, 2007: 54). Nesse caso, a máquina de execução cria e lança, para cada ação, um objeto da classe `Thread`. Assim, a máquina de apresentação não precisa se preocupar em como essa classe é implementada, sendo sua única exigência que a classe `Thread` ofereça pelo menos algum nível de concorrência.

O único ponto da implementação original que teve que ser modificado com o objetivo de se remover o uso das `threads` POSIX foi, portanto, a implementação da classe `Thread`. No Ginga-NCL para dispositivos portáteis, a classe abstrata `Thread` é, na verdade, um Background Active Object. Essa substituição foi o suficiente para fazer com que as aplicações fossem executadas com maior velocidade. O sincronismo se tornou, então, possível, e aplicações até dez vezes maiores puderam ser desenvolvidas. Ressalta-se que, embora o Background Active Object tenha sido utilizado, em nenhum caso houve a

necessidade de se dividir o procedimento implementado no método `RunL` em mais de uma parte. Ou seja, todos os procedimentos eram leves o suficiente para serem executados de uma só vez.

Existe, porém, o caso do tratamento de âncoras temporais, que também faz uso da classe `Thread`, onde a substituição da implementação das `threads` POSIX pelo Background Active Object introduziu, nesse caso específico, um erro. Esse problema é apresentado juntamente com a sua solução na próxima seção.

3.6. O Tratamento de Âncoras Temporais

A substituição das `threads` POSIX pelos Active Objects introduziu um erro no tratamento das âncoras temporais.

Para entender como esse erro foi gerado e a sua solução, é necessário, antes, introduzir o conceito de âncoras temporais. Também é importante apresentar como foi feita a sua implementação na versão do Ginga-NCL para dispositivos fixos.

Uma âncora temporal define um intervalo de tempo dentro da duração da apresentação de um elemento de mídia qualquer. Para criar uma âncora temporal, usa-se o elemento <area> — como em toda a criação de ancoras — e especifica-se os seus atributos “begin” e “end”. Esses atributos representam, respectivamente, o início e o fim do intervalo relativos ao início da apresentação do conteúdo de mídia. Essa tag deve ser definida como um elemento do nó de mídia em que se quer definir um intervalo temporal. Como exemplo, a Figura 6 ilustra um cenário onde várias âncoras temporais são definidas.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ncl id="ExemploAncoraTemporal.ncl"
xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">

   <head>

      <regionBase>
         <region id="rgVideo" left="0%" top="0%"
                width="50%" height="50%"/>

         <region id="rgImagem" left="50%" top="0%"
                width="50%" height="50%"/>
      </regionBase>

      <descriptorBase>
         <descriptor id="dVideo" region="rgVideo" />
         <descriptor id="dImagem" region="rgImagem" />
      </descriptorBase>

      <connectorBase>
         <causalConnector id="onBeginStart">
                <simpleCondition role="onBegin"/>
                <simpleAction role="start" max="unbounded" qualifier="par"/>
         </causalConnector>
      </connectorBase>

   </head>

   <body>

        <port id="pInicio1" component="video"/>

        <media type="video/3gp" id="video"
                src="whatisthematrix.3gp" descriptor="dVideo">
                <area id="intervalo1" begin="5s" end="10s"/>
                <area id="intervalo2" begin="11s" end="13s"/>
                <area id="intervalo3" begin="15s" end="20s"/>
        </media>

        <media type="image/mbm" id="imagem1" src="matrix.mbm" descriptor="dImagem"/>

        <link id="lnk1" xconnector="onBeginStart">
                <bind component="video" interface="intervalo1" role="onBegin" />
                <bind component="imagem1" role="start" />
        </link>

   </body>
</ncl>
```

Figura 6: Exemplo de um documento NCL com âncoras temporais.

Um elemento <area> definido em um nó de mídia (<media>) pode ser referenciado como uma interface do objeto de mídia, como exemplifica o link lnk1 da Figura 6. Âncoras temporais podem, assim, serem usadas com o objetivo de se fazer sincronismo temporal entre trechos de mídias. Na Figura 6, quando o vídeo definido tiver completado 5 segundos, uma transição de estado de evento de ”start”, associado à sua primeira âncora, ocorre, indicando o início do intervalo. Analogamente, quando o vídeo alcançar os dez segundos, uma transição de estado de evento de ”stop”, associada a mesma âncora, ocorre, indicando o fim do intervalo. A máquina de apresentação capta essas transições de estados de eventos e, se houver uma ação associada a uma dessas transições, essa ação é executada. Como exemplo, no link lnk1 da Figura 6, é

especificado que a apresentação da imagem1 deve ser iniciada quando a primeira âncora definida para o vídeo começar sua exibição. A máquina de apresentação capta a transição de estado de evento de início da âncora e inicia a apresentação da imagem1, conforme especificado.

Para implementar uma âncora temporal, precisa-se, basicamente, contabilizar o tempo decorrido da exibição da mídia, lançando as transições de estados de eventos “start” e “stop” nos momentos definidos pelas suas âncoras. Na implementação de referência do Ginga-NCL para dispositivos fixos, a máquina de apresentação dispara, para cada mídia possuidora de âncoras temporais, uma `thread` separada. Essa `thread` fica responsável por organizar uma fila crescente com os tempos definidos pelos elementos <area> associados. Essa fila é usada para descobrir os momentos em que as transições de estados de eventos devem ser disparadas. A Figura 7 ilustra como fica, simplificadamente, a fila do exemplo apresentado na Figura 6:

5s	10s	11s	13s	15s	20s
begin	end	begin	end	begin	end

Figura 7: Exemplo de fila de tempos definidos em âncoras temporais.

De posse dessa lista, resta à `thread` contabilizar o tempo da mídia e lançar as transições de estados de eventos nos tempos definidos pela fila. A contagem do tempo decorrido das mídias é sempre feita nos próprios objetos de exibição. No caso de mídias com tempo de duração pré-estabelecido, como as de vídeo ou de áudio, suas API’s de exibição normalmente possuem uma função que retorna o tempo decorrido de apresentação. Quando as mídias não possuem tempo de duração definido, ou quando as suas API de exibição não fornecem uma função que retorne o tempo decorrido de apresentação, a implementação dos seus exibidores deve, obrigatoriamente, criar um contador próprio. Com a fila criada e com a informação do tempo decorrido de apresentação da mídia, a `thread` realiza o seguinte procedimento de quatro passos:

1- Remove o primeiro elemento da fila;
2- Calcula a diferença entre o tempo do elemento removido da fila e o tempo corrente da mídia. Essa diferença é exatamente o tempo que será necessário aguardar até que o tempo definido por uma das âncoras seja alcançado;
3- Usa diferença encontrada em um comando de sleep.

4- Acorda do comando de sleep. Nesse momento, o tempo definido por uma das âncoras foi alcançado. A implementação lança, portanto, uma transição de estado de evento de start ou stop dependendo se o tempo alcançado foi de início ou término de uma âncora;

Esse procedimento se repete até que a fila tenha se esvaziado. Isso basta para que o uso de âncoras temporais seja tratado corretamente. Na implementação apresentada nesta dissertação, no entanto, o uso das `threads`, como já foi apresentado em seções anteriores, é proibitivo. No lugar delas, foi feito o uso dos Active Objects, que executam todos na `thread` principal do processo associado a aplicação Ginga-NCL.

Pelo fato dos Actives Objects se encontrarem em uma mesma `thread`, a chamada a uma função de sleep dentro das funções `RunL` dos Actives Objects colocaria todo o processo da aplicação Ginga-NCL em espera, impedindo que outras ações pudessem ser feitas. O tratamento das âncoras temporais tem, então, de ser feito sem a necessidade de bloquear o Active Object e, conseqüentemente, a aplicação como um todo.

A solução para esse problema foi usar um Active Object especial, o `CTimer` (Symbian Ltd, Nokia, 2006), e implementá-lo como um Background Active Object. `CTimer` é como um Active Object normal, a não ser pelo fato dele já possuir métodos que encapsulam funções assíncronas de contagem de tempo. O mais importante deles é o `After`, que recebe como parâmetro um tempo a ser aguardado. Quando chamado, esse método chama uma função assíncrona de sistema passando no seu parâmetro o tempo a ser aguardado. Essa função, por sua vez, executa em paralelo ao Active Object e aguarda pela passagem do tempo especificado no seu parâmetro. Quando o tempo termina, a função notifica o fato, sinalizando o semáforo do Active Scheduler. Isso faz com que, em algum determinado momento, o método `RunL` do Active Object em questão seja chamado pelo Active Scheduler.

A solução encontrada para o tratamento das âncoras temporais foi, portanto, realizar a sua implementação no método `RunL` de um `CTimer`. Para fazer essa implementação, foi preciso analisar o procedimento de quatro passos necessário para o tratamento de âncoras temporais. Se observado com cuidado, esse procedimento pode ser dividido em duas partes: A primeira (1) compreende os passos 1, 2 e 3 do procedimento, sendo que, no passo 3, o método `After` do `CAtive` deve ser usado ao invés do sleep. A segunda parte (2) compreende o

passo 4, que ocorre logo depois que o tempo de espera tiver terminado. Essas duas partes foram implementadas separadamente no método `RunL` do `CTimer`, que controla o procedimento de âncoras temporais.

Para iniciar a execução do método `RunL`, um outro método teve que ser implementado com o objetivo de fazer com que o `CTimer` ficasse elegível pelo Active Scheduler, como descrito na Seção 3.5. Ao fazer a chamada desse método, o `CTimer` fica elegível pelo Active Scheduler e, em algum determinado momento, o método `RunL` do `CTimer` é chamado. Quando a função `RunL` é finalmente chamada, somente (1) é executado. Assim que o tempo de espera termina, `RunL` é novamente escalonado e, dessa vez, (2) é executado com o objetivo de lançar o evento de “start” ou de “stop” da ancora temporal. Se existirem mais elementos na fila de tempos definidos pelas ancoras temporais de uma mídia, (1) é chamado novamente, e o ciclo é reiniciado.

Implementado dessa forma, o tratamento de âncoras temporais na implementação do Ginga-NCL para dispositivos portáteis foi minimamente alterado, mantendo as características base da implementação para dispositivos fixos, com a vantagem que o uso de `threads` foi completamente removido, contribuindo para o funcionamento adequado e otimizado da nova implementação.

3.7. Exibidores no Symbian C++

Os exibidores são os elementos responsáveis pela apresentação do conteúdo dos objetos de mídia. Os exibidores dependem da plataforma de desenvolvimento e, portanto, representam a parcela com mais problemas de portabilidade do middleware Ginga-NCL.

Para facilitar o procedimento de porte do Ginga-NCL, uma camada de abstração de exibidores foi desenvolvida desde sua implementação de referência para dispositivos fixos. Essa camada implementa uma série de funções que devem ser mapeadas nas API’s específicas dos exibidores da plataforma implementada. A máquina de apresentação precisa acessar apenas essa camada a fim de controlar a execução de um exibidor, não tendo a necessidade de se preocupar com as características específicas da plataforma de desenvolvimento.

No controle da execução dos exibidores, a máquina de apresentação precisa associar os objetos de exibição às regiões especificadas pelo documento NCL. Esse procedimento, em si, exige o acesso ao display do dispositivo, o que também é um procedimento específico de plataforma. Pelo mesmos motivos do caso dos exibidores, uma camada de abstração para o acesso ao display do dispositivo foi definida. De forma similar, suas funções devem ser mapeadas em API's específicas da plataforma. A máquina de apresentação usa essa camada para gerenciar as regiões da aplicação NCL e, assim como no caso dos exibidores, não precisa se preocupar com as características específicas da plataforma de desenvolvimento.

Em uma apresentação de uma aplicação NCL, a máquina de apresentação cria, primeiro, as regiões definidas no documento NCL. Depois, ao longo da apresentação da aplicação, solicita ao Gerenciador de Exibidores, visto na Seção 3.1, a instanciação dos elementos de mídia e os associa às regiões previamente criadas.

A máquina de apresentação também pode acessar os exibidores através da sua camada de abstração. Isso é necessário para, por exemplo, iniciar, pausar ou parar um vídeo, aumentar ou diminuir o som de um áudio, dentre outras ações desse tipo. A Figura 8 ilustra o procedimento que acaba de ser descrito, levando em conta as camadas de abstração mencionadas.

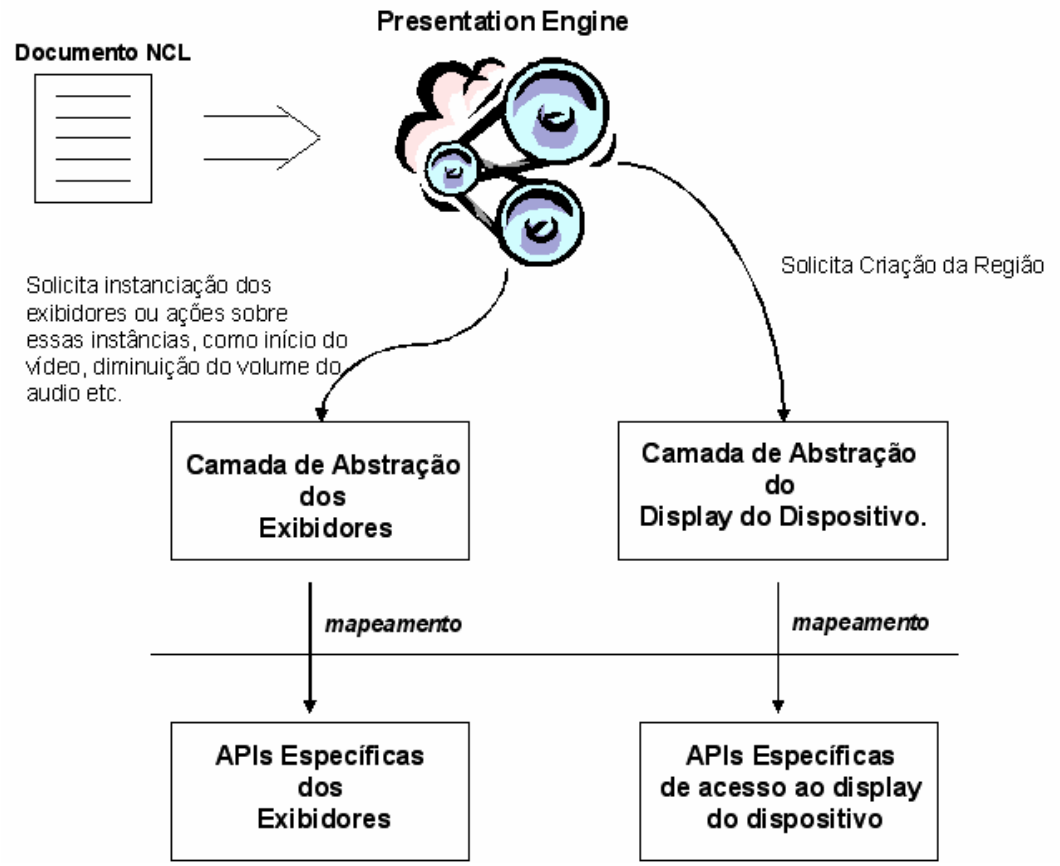

Figura 8: Acesso das camadas de abstração dos exibidores e do Display do dispositivo por parte da máquina de apresentação.

O procedimento, como um todo, é independente de plataforma, e não necessitou sofrer alterações. O que precisou ser feito na implementação corrente foi o mapeamento das camadas de abstração de display e de exibidores para as API's específicas do Symbian.

Em Symbian, o acesso ao display é feito através de objetos chamados de Window. Entretanto, a apresentação de mídias na tela normalmente não é feita através desse recurso. Ao invés dele, usam-se os Controls, já que o uso direto de uma Window consome muitos recursos e é muito complexo. Os Controls, por sua vez, abstraem o uso das API's providas pela Window e compartilham esse recurso entre si, ou seja, dois ou mais Controls podem fazer uso de uma mesma Window. Normalmente, uma aplicação Symbian possui apenas uma Window associada, pois a criação de mais objetos desse tipo causaria uma degradação do sistema. Os Controls, portanto, são usados para implementar as diferentes interfaces de uma aplicação e normalmente compartilham uma única Window. Por exemplo, um Control poderia ser responsável por mostrar uma lista de opções na tela e, um outro, por apresentar uma imagem de fundo. Ambos podem compartilhar a mesma Window, economizando, assim, recursos.

Para implementar um Control, é preciso estender a classe `CCoeControl` (Symbian Ltd, Nokia, 2006) implementando alguns dos seus métodos abstratos. O mais importante deles é o `Draw`, que é usado para aplicar alterações na tela do dispositivo. Quando um Control é iniciado ou quando o seu método `DrawNow` é utilizado, a função `Draw` é chamada. Dentro da função `Draw`, o objeto `CWindowGc` (Symbian Ltd, Nokia, 2006), que pode ser obtido através da chamada a função `SystemGc` pertencente ao Control, deve ser usado para fazer o desenho na tela. Nesse objeto estão encapsuladas todas as funções de desenho, como definição de cor, estilo do pincel, criação de formas geométricas, dentre outras. A apresentação de objetos no display de qualquer dispositivo com o Symbian OS instalado é feita, portanto, na função `Draw` dos Controls da aplicação por meio do objeto `CWindowGc`. Quando a máquina de apresentação associa, na implementação proposta, um exibidor a uma região NCL, estará, na verdade, associando-o a um Control criado para aquela região.

Uma limitação que pode ser citada na forma como é feito o gerenciamento gráfico no Symbian, é a existência de apenas uma camada gráfica de apresentação. Isso significa que imagens e vídeos, ou quaisquer outros objetos de exibição, serão sempre apresentados em uma mesma camada. Uma diferença da implementação aqui exposta em comparação à implementação de referência para dispositivos fixos é que, para dispositivos portáteis, nenhum objeto pode ser apresentado por cima de um vídeo. Essa limitação ocorre justamente por causa da limitação da existência de uma única camada.

Entre os vários exibidores de tipos de mídias diferentes, já foram implementados os de imagem estática, áudio, vídeo e HTML.

Para apresentar uma imagem na tela do dispositivo, é preciso usar, dentro do método `Draw` de um Control, o método `DrawBitmap` pertencente à classe `CWindowGc`. Esse método recebe como parâmetro a posição onde a imagem deve ser desenhada, além de um objeto do tipo `CFbsBitmap`, que é usado para carregar, através do seu método `Load`, imagens estáticas armazenadas dentro de um buffer ou de algum arquivo em disco. O `CFbsBitmap`, entretanto, só é capaz de carregar imagens do tipo mbm (Multi-BitMaps), que é um tipo específico de imagem usada na plataforma Symbian.

O vídeo e o áudio são providos com o uso da Multi Media Framework (MMF), onde são definidas as classes `CVideoPlayerUtility` e `CMdaAudioPlayerUtility`, usadas, respectivamente, para tocar vídeos e áudios. O funcionamento dessas duas classes é muito similar. Ambas oferecem funções idênticas capazes de carregar um vídeo ou um áudio de origens

diferentes. Um exemplo é a função `OpenFileL`, que abre um vídeo ou áudio de uma arquivo salvo em disco. Caso a abertura do arquivo seja bem sucedida, o método Play das classes `CVideoPlayerUtility` e `CMdaAudioPlayerUtility` pode ser usado para começar a tocar o objeto de mídia. Existem outras funções nessas classes que podem ser usadas na manipulação das propriedades do vídeo ou do áudio. O vídeo, especificamente, pode ter as suas dimensões modificadas pelo seu Control associado, ou seja, o Control pode fazer, quando necessário, uso das funções de redimensionamento providas pelo `CVideoPlayerUtility`. O áudio, por outro lado, não possui características de visualização e, portanto, não precisaria ter um Control associado, mas essa associação foi feita mesmo assim com o objetivo de manter a padronização na implementação.

Os dispositivos Symbian possuem uma lista de tipos de áudio e vídeo que possuem plug-ins associados. As classes `CVideoPlayerUtility` e `CMdaAudioPlayerUtility`, quando tentam carregar um objeto de áudio ou de vídeo, acessam essa lista para descobrir se o dispositivo possui o plug-in adequado para tocar o elemento que está sendo carregado. Se o dispositivo possuir o plug-in apropriado, quando o comando Play for executado, as classes `CVideoPlayerUtility` e `CMdaAudioPlayerUtility` associarão o objeto de mídia carregado a esse plug-in, e começarão a tocá-lo. Dessa forma, essas classes abstraem os tipos de áudio e vídeo que podem ser tocados, ou seja, elas podem ser usadas para tocar qualquer tipo de mídia de áudio ou vídeo, bastando apenas que o dispositivo possua os plug-ins necessários.

Finalmente, o exibidor HTML também foi implementado. Não existe API genérica para a apresentação de documentos HTML no Symbian, portanto foi necessário fazer uso de uma API específica da plataforma S60. Isso significa que o exibidor HTML funcionará apenas em dispositivos específicos dessa plataforma. A classe usada foi a `CBrCtlInterface`, sendo o mais importante dos seus métodos o `LoadUrlL`, que carrega uma página HTML de uma URL qualquer. Para criar um objeto da classe `CBrCtlInterface` é necessário usar a função estática `CreateBrowserControlL`, passando o Control associado a esse exibidor em um dos seus parâmetros. Qualquer alteração nesse Control vai, a partir de então, refletir no exibidor HTML.

A existência das camadas de abstração descritas facilitou muito o procedimento de porte dos exibidores, que se tornou algo bem simples e direto. O maior problema encontrado na implementação dos exibidores foi que, apesar de existir documentação para as API’s específicas Symbian, a informação

provida normalmente é muito sucinta e, em alguns casos, insuficiente. Os exemplos encontrados também são muito complexos e acabavam dificultando ainda mais o procedimento de aprendizagem.

3.8. Considerações Finais

Durante a implementação do Ginga-NCL para dispositivos portáteis, ficou clara a importância que um sistema aberto e padronizado tem no processo de desenvolvimento.

A plataforma nativa do Symbian, e seu ambiente de desenvolvimento Symbian C++, foi projetada especificamente para os dispositivos portáteis e é muito particular. As API's de desenvolvimento Symbian C++, por exemplo, guardam poucas semelhanças com as API's padrões de desenvolvimento C++, linguagem da qual Symbian C++ se baseia. Isso dificulta qualquer procedimento de porte e força os programadores a aprenderem um novo conjunto de API's de desenvolvimento bem diferente daqueles que estão acostumados. Dessa forma, o procedimento de desenvolvimento torna-se muito complexo e pouco atrativo para a plataforma em questão. O fato da plataforma Symbian ser aberta contribui para a observância desses problemas de padronização e especificidade da plataforma Symbian, e, principalmente, para sua melhora.

Com o apoio da comunidade, a Symbian Ltd, criadora do sistema operacional Symbian, foi capaz de perceber a importância da padronização de uma plataforma e ofereceu o P.I.P.S.. Mais tarde, a Nokia, que desenvolve a API específica S60, disponibilizou o Open C, o que contribuiu ainda mais para a padronização e, conseqüentemente, para o porte de aplicações desenvolvidas em outras plataformas. A falta da biblioteca STL, usada comumente em programas desenvolvidos em C++, também prejudicava o procedimento de porte. Mais uma vez, por ser uma plataforma aberta, um integrante da comunidade Symbian pôde fazer, com a ajuda da já disponibilizada P.I.P.S., o seu próprio porte de uma implementação STL para o Symbian OS.

Os esforços em torno da disponibilização das bibliotecas P.I.P.S, Open C e STL contribuem muito para o desenvolvimento e o porte de aplicações para a plataforma Symbian, e, conseqüentemente, contribuíram muito para o porte e o desenvolvimento do Ginga-NCL para dispositivos portáteis. No início da implementação, essas bibliotecas não foram utilizadas, como é descrito na Seção 3.2. A opção foi implementar mapeamentos das API's utilizadas na

implementação de referência para dispositivos fixos em API's específicas da plataforma Symbian, o que se demonstrou ser muito complexo e demorado. Quando as bibliotecas P.I.P.S., Open C e STL passaram a ser utilizadas, o desenvolvimento tornou-se mais simplificado. Sem essas bibliotecas, o desenvolvimento completo do Ginga-NCL para dispositivos portáteis seria, sem dúvida, muito mais complicado além do que demoraria muito mais tempo.

Em outros sistemas abertos, como o Linux, espera-se ter uma facilidade de porte e de desenvolvimento igual ou talvez até mesmo maior do que no Symbian. Por outro lado, sistemas de plataforma fechada, como o BlackBerry, possuem sua própria plataforma de desenvolvimento e provavelmente não devem oferecer suporte à maioria das bibliotecas padrão. Isso certamente dificultaria muito o porte e desenvolvimento do Ginga-NCL, como de qualquer outra aplicação.

Outra questão que deve ser mencionada é a alteração do módulo Conversor na implementação do Ginga-NCL para dispositivos portáteis. Essa tarefa consumiu muito tempo, uma vez que praticamente todo o módulo em questão teve que ser re-implementado, mantendo-se inalteradas somente as suas interfaces com o resto da arquitetura Ginga-NCL. A experiência obtida com a implementação do módulo Conversor evidenciou que não seria necessário fazer uso dos recursos oferecidos pelo modelo DOM. Normalmente, o parser DOM é usado quando a estrutura do documento XML é muito complexa, com muitos níveis e muitas referências entre objetos de níveis diferentes, quando é necessário fazer uso pontual da estrutura XML em diversos momentos da execução do programa, quando é preciso acessar elementos diferentes da estrutura XML em um mesmo momento, ou quando é necessário alterar o documento XML.

Foi possível avaliar, entretanto, que a NCL em si não é suficientemente complexa a ponto de exigir o uso do DOM na implementação do módulo Conversor, que foi re-projetado utilizando a API SAX na versão para dispositivos portáteis. Documentos NCL normalmente não possuem tantos níveis e as referências entre elementos podem ser resolvidas sem muitas dificuldades. Portanto, nenhuma característica dessa linguagem impede, aparentemente, o uso do SAX, ou exige que o DOM seja usado. Outro ponto que deve ser observado é que também não há a necessidade de fazer acessos repetidos à estrutura do documento NCL. No Ginga NCL, o documento NCL recebido da emissora ou de qualquer outra fonte deve ser convertido em uma estrutura de dados baseada no modelo NCM. A partir de então, somente essa estrutura é

usada, sendo desnecessário acessar novamente o documento NCL e, conseqüentemente, manter a estrutura montada por um parser DOM.

Existe, entretanto, um recurso do Ginga-NCL que lida com elementos da linguagem NCL e que não foi implementado na versão desse middleware para dispositivos portáteis. Esse recurso é o tratamento de edição ao vivo de aplicações NCL. O Ginga-NCL permite que sejam feitas edições ao vivo em qualquer aplicação NCL que esteja sendo apresentada. Para fazer uma edição, um documento NCL específico deve ser entregue ao Ginga-NCL, que, por sua vez, o processa e aplica as alterações na estrutura NCM da aplicação que está sendo apresentada. Dessa forma, igualmente para o caso das edições, o uso do DOM parser também não se faz necessário. Sem a necessidade de uso do parser DOM, ficou evidenciado que o SAX demonstra-se muito mais vantajoso pela sua eficiência e pelo seu baixo consumo de memória, não só em dispositivos portáteis, mas também em terminais fixos.

Uma outra questão interessante que deve ser salientada diz respeito ao uso das `threads` na implementação para dispositivos portáteis, o que demonstrou claramente o problema das limitações de recursos encontradas nesses dispositivos. Como foi apresentado nesta dissertação, o uso de `threads` POSIX na implementação do Ginga-NCL para dispositivos portáteis tornou-se proibitivo, fazendo com que a execução das aplicações NCL ficasse extremamente lenta. Para contornar o problema, as `threads` foram substituídas pelos Active Objects, permitindo a execução das aplicações de forma correta e eficiente, mesmo sem o paralelismo das `threads`. Em outras plataformas, a possibilidade do uso das `threads` vai depender de como esse recurso é implementado e de como é o seu consumo de recursos. Todavia, se o uso de `threads` em qualquer outra plataforma também for proibitivo, é possível definir outras formas de escalonamento que viabilizem a execução do Ginga-NCL, como já foi demonstrado com o uso dos Active Objects no Symbian.

A implementação dos exibidores, por sua vez, é muito dependente de plataforma. O fato do Ginga-NCL possuir uma abstração do uso desses elementos facilitou muito a implementação, algo que certamente deve se repetir na implementação do Ginga-NCL em outras plataformas. Em Symbian, os exibidores de vídeo e de HTML foram os mais complicados de se implementar. O exibidor de texto certamente é o mais problemático de todos, tanto que não foi possível, ainda, implementá-lo na versão atual do Ginga-NCL para dispositivos portáteis. Por conta disso, a sua implementação é citada nos trabalhos futuros

desta dissertação. A implementação dos exibidores em outras plataformas depende muito de como são as suas API's específicas de exibição.

A implementação do Ginga-NCL para dispositivos portáteis resultante do trabalho apresentado nesta dissertação foi testada em um emulador de dispositivo Symbian S60 e em dois dispositivos portáteis. Os testes são detalhados no Capítulo 4.

Em relação à implementação do Ginga-NCL para dispositivos fixos, algumas limitações devem ser observadas. A primeira delas é que não é possível apresentar uma imagem sobre um vídeo por causa da existência de uma única camada gráfica de apresentação, como é descrito na Seção 3.7. A segunda limitação que pode ser citada, é a impossibilidade de se definir níveis de transparência para os vídeos. Como última limitação, pode-se citar que a soma total dos tamanhos das mídias apresentadas ao mesmo tempo em uma aplicação NCL não pode ser muito alta. O limite para isso depende muito do dispositivo em que a aplicação NCL está executando, mas, nos testes realizados, somas maiores que 2Mb causaram o término da aplicação por falta de recursos.

Como é descrito no Capítulo 4, o funcionamento do Ginga-NCL foi testado em diferentes aplicações NCL, sendo que o funcionamento delas ocorreu de forma idêntica no emulador e nos dispositivos portáteis, e não foi observada qualquer degradação do sistema. As aplicações NCL executaram sem aparente perda de eficiência e o sincronismo funcionou adequadamente. Dessa forma, pode-se concluir que a implementação apresentada nessa dissertação foi bem sucedida.

As ferramentas necessárias para a compilação do Ginga-NCL, o emulador e os dispositivos usados, bem como o procedimento de instalação do Ginga-NCL, são apresentados no Apêndice A desta dissertação.