Bruno Cavalcanti Muniz

Notificação e Controle de Versões para o Suporte à Autoria Cooperativa no Sistema HyperProp

DISSERTAÇÃO DE MESTRADO

Departamento de Informática

Rio de Janeiro, 28 de Agosto de 2000

Pontifícia Universidade Católica do Rio de Janeiro

Bruno Cavalcanti Muniz

Notificação e Controle de Versões para o Suporte à Autoria Cooperativa no Sistema HyperProp

Dissertação de Mestrado apresentada ao Departamento de Informática da PUC-Rio, como parte dos requisitos para obtenção do título de Mestre em Informática: Ciência da Computação. Orientador: Luiz Fernando Gomes Soares

Departamento de Informática

Pontifícia Universidade Católica do Rio de Janeiro

Rio de Janeiro, 28 de Agosto de 2000

Este trabalho é dedicado

Aos meus pais e irmãos, que tanto amo.

Agradecimentos

- Ao Prof. Luiz Fernando, por sua incansável e valiosa orientação.
- Aos amigos do TeleMídia, em especial Débora e Rogério, grandes companheiros de trabalho.
- Aos amigos das Repúblicas (Farinha e Dragão do Mar, em diferentes versões) pelo alto astral sempre presente.
- Aos professores da UECE Marcos Negreiros, Wamberto Vasconcelos e Fernando Carvalho pela contribuição em minha formação acadêmica.
- A CAPES e Embratel pelo apoio financeiro.
- Ao extraordinário Johann Sebastian Bach, homem de grande humildade (superada somente pela sua imensa genialidade musical), pelos infinitos prazeres auditivos.
- Ao Ser Supremo, pela criação do universo, intrigante por definição.

Resumo

É importante que um sistema de autoria hipermídia ofereça suporte à edição simultânea de documentos, realizada por vários autores, de maneira adequada e eficiente. Para apoiar esse processo, sistemas hipermídia incorporam mecanismos de controle de versões integrados a esquemas de notificação, que informam os autores sobre eventos significativos relacionados à criação e edição de versões.

Esta dissertação especifica mecanismos de controle de notificação para o Modelo de Contextos Aninhados (NCM), implementados e acoplados ao sistema de autoria hipermídia HyperProp. É importante destacar que esses mecanismos podem ser adaptados para outros modelos hipermídia baseados em composições e que ofereçam funcionalidades para controle de versões. Adicionalmente, é apresentada uma implementação do controle de versões do NCM que possibilita, entre outras facilidades, a visualização gráfica do histórico de hiperdocumentos no sistema HyperProp.

Palavras-chave: Notificação, Versionamento Hipermídia, Autoria Cooperativa Semi-Síncrona, NCM

Abstract

It is important to a hypermedia authoring system to support simultaneous document editing, done by several authors, in an adequate and efficient way. To accomplish that, hypermedia systems incorporate version control mechanisms integrated to notification schemes, which inform authors about significant events related to creation and editing of versions.

This work specifies notification control mechanisms for the Nested Context Model (NCM), which were implemented within the hypermedia authoring system HyperProp. It is important to observe that these mechanisms can be adapted to other hypermedia models based on compositions and offering version control functionalities. In addition, an implementation of the NCM version control is also presented, enabling, among other facilities, the graphical visualization of hyperdocuments history in the HyperProp system.

Keywords: Notification, Hypermedia Versioning, Semi-Synchronous Cooperative Authoring, NCM

Conteúdo

Lista de Figuras

1. Introdução

É um fato bem conhecido que documentos são criados frequentemente por um grupo de autores [HAAKE 1992]. No caso de hiperdocumentos grandes, esse fato torna-se ainda mais comum. Portanto, é de grande utilidade que um sistema hipermídia ofereça recursos para que vários autores possam contribuir na autoria de um documento hipermídia adequada e eficientemente.

O processo onde vários autores criam e editam um determinado documento, potencialmente ao mesmo tempo e utilizando uma rede de computadores, é denominado de *autoria cooperativa* [REINERT 1998]. O suporte à autoria cooperativa em sistemas hipermídia é uma necessidade que, além de ser empiricamente comprovada, já foi apontada em vários trabalhos da literatura [HALASZ 1988] [WIIL 1993a] [HAAKE 1998] [WEBDAV 2000].

Para apoiar esse processo, sistemas de autoria hipermídia incorporam mecanismos de controle de versões integrados a esquemas de notificação de alteração. Os sistemas de controle de versões permitem a criação, gerenciamento e navegação de estados significativos de um objeto, enquanto que o controle de notificação informa os autores sobre eventos significativos relacionados à edição, como a criação de novas versões de um determinado nó, atualização de atributos e obsolescência de nós. Com esse mecanismo, o estado de edição de um hiperdocumento pode ser divulgado para todos os autores.

O sistema hipermídia HyperProp [RODRIGUES 1998], desenvolvido pelo Laboratório TeleMídia do Departamento de Informática da PUC-Rio, é uma ferramenta que tem como objetivo o tratamento de hiperdocumentos, fornecendo meios para autoria e apresentação de documentos hipermídia com sincronismo temporal e espacial. O sistema, que segue uma implementação cliente-servidor, é baseado no modelo conceitual de dados hipermídia, o Modelo de Contextos Aninhados (*Nested Context Model* - NCM) [SOARES 1995], que representa os conceitos estruturais dos dados, os eventos e os relacionamentos entre os dados e também define

regras de estruturação e operações sobre os dados para manipulação e atualização das estruturas.

Portanto, é importante ressaltar que a notificação no versionamento dos dados no NCM, para o suporte à autoria cooperativa no sistema HyperProp, é o foco principal deste trabalho.

1.1 Objetivos

Um dos objetivos deste trabalho consiste em implementar o controle de versões do NCM, de acordo com especificações mais recentes do modelo [SOARES 2000]. A implementação será acoplada ao sistema HyperProp, que oferecerá as funcionalidades de versionamento por meio da interface gráfica de *browsers* do sistema [PINTO 2000]. Dentre algumas facilidades oferecidas, pode-se citar a possibilidade de visualização gráfica do histórico de derivação de um objeto.

Conforme comentado anteriormente, o objetivo principal deste trabalho é apresentar um modelo de notificação original para o NCM, e implementá-lo. O mecanismo de notificação atuará junto ao controle de versões, oferecendo suporte eficiente à autoria semi-síncrona de documentos hipermídia. A *autoria semi-síncrona* ocorre quando diversos autores atuam em um dado objeto compartilhado ao mesmo tempo, porém em sessões individuais [MINOR 1993] [WIIL 1998]. Normalmente, os autores trabalham com versões do objeto nessas sessões individuais.

A implementação do mecanismo de notificação também será acoplada ao sistema HyperProp, que possibilitará a manipulação de aspectos do controle de notificação por meio da interface gráfica do sistema.

1.2 Organização da Dissertação

A descrição do NCM e de seu mecanismo para controle de versões, especificado em detalhes em [SOARES 2000], será apresentada no Capítulo 2. Os pontos relevantes nas operações de versionamento que indicam onde o controle de notificação deverá atuar serão ressaltados.

O Capítulo 3 apresenta um modelo de notificação original para o NCM, que tem como objetivos principais a extensibilidade, adaptabilidade e expressividade. Será visto também que, apesar de especificado inicialmente para o NCM, o modelo de notificação pode ser aplicado a outros modelos hipermídia que sejam baseado em composições e que ofereçam mecanismos para controle de versões.

O Capítulo 4 detalha a implementação do controle de versões e do mecanismo de notificação proposto, juntamente com a incorporação dessas funcionalidades ao sistema HyperProp. Ainda nesse capítulo, será mostrado como a notificação no versionamento de nós no NCM pode ser utilizada como base para diferentes procedimentos relacionados à edição cooperativa. Algumas estratégias reativas, que provocam ações no sistema de autoria no recebimento de notificações, serão sugeridas.

O Capítulo 5 discute trabalhos relacionados à versões e notificação para suporte à autoria cooperativa em sistemas hipermídia. Questões importantes pertinentes a esses pontos serão salientadas em cada um dos trabalhos discutidos, em comparação com o conteúdo apresentado nos capítulos 2, 3 e 4.

O Capítulo 6, reservado para as conclusões, inclui as contribuições e sugestões de trabalhos futuros, finalizando esta dissertação.

2. Controle de Versões

2.1 Introdução

Considera-se *versão*, uma instância de um objeto registrada em um dado instante, e que está associada a um histórico de derivação, isto é, versões de um mesmo objeto são associadas por meio de relações de predecessão e sucessão.

O mecanismo de controle de versões, que permite a navegação e o acesso à estados anteriores de um objeto, têm sido bastante utilizado em várias áreas como Banco de Dados, Engenharia de Software, CAD (*Computer Aided Design*), CSCW (*Computer-Supported Cooperative Work*) e Hipermídia.

O objetivo deste capítulo é apresentar o controle de versões do NCM. A descrição do NCM, necessária para a compreensão deste e dos próximos capítulos, será apresentada na Seção 2.2. Seu mecanismo para controle de versões será descrito na Seção 2.3, sempre ressaltando os casos em que o controle de notificação, a ser visto no Capítulo 3, deve agir. Ainda nessa seção, algumas contribuições dadas por essa dissertação e que foram adotadas no modelo NCM serão ressaltadas.

2.2 O NCM

O Modelo de Contextos Aninhados (*Nested Context Model* - NCM) [SOARES 2000] é um modelo conceitual de dados hipermídia, orientado a objetos, cujo modelo de interface separa os componentes de dados e de exibição dos objetos, permitindo a manipulação de documentos independente da plataforma final de exibição.

Original na definição de composições aninhadas [CASANOVA 1991], a definição de documentos hipermídia no NCM é baseada nos conceitos usuais de nós e elos. *Nós* são fragmentos de informação e *elos* são usados para a definição de relacionamentos entre os nós que interligam.

O modelo distingue duas classes básicas de nós, chamados de nós terminais (ou de conteúdo) que armazenam os dados (texto, áudio, vídeo, ...) propriamente ditos, e nós de composição, que proporcionam a estruturação lógica de um documento.

A Figura 1 apresenta a hierarquia de classes atual do NCM. As classes que são relevantes para esta dissertação serão definidas ao longo desta seção.

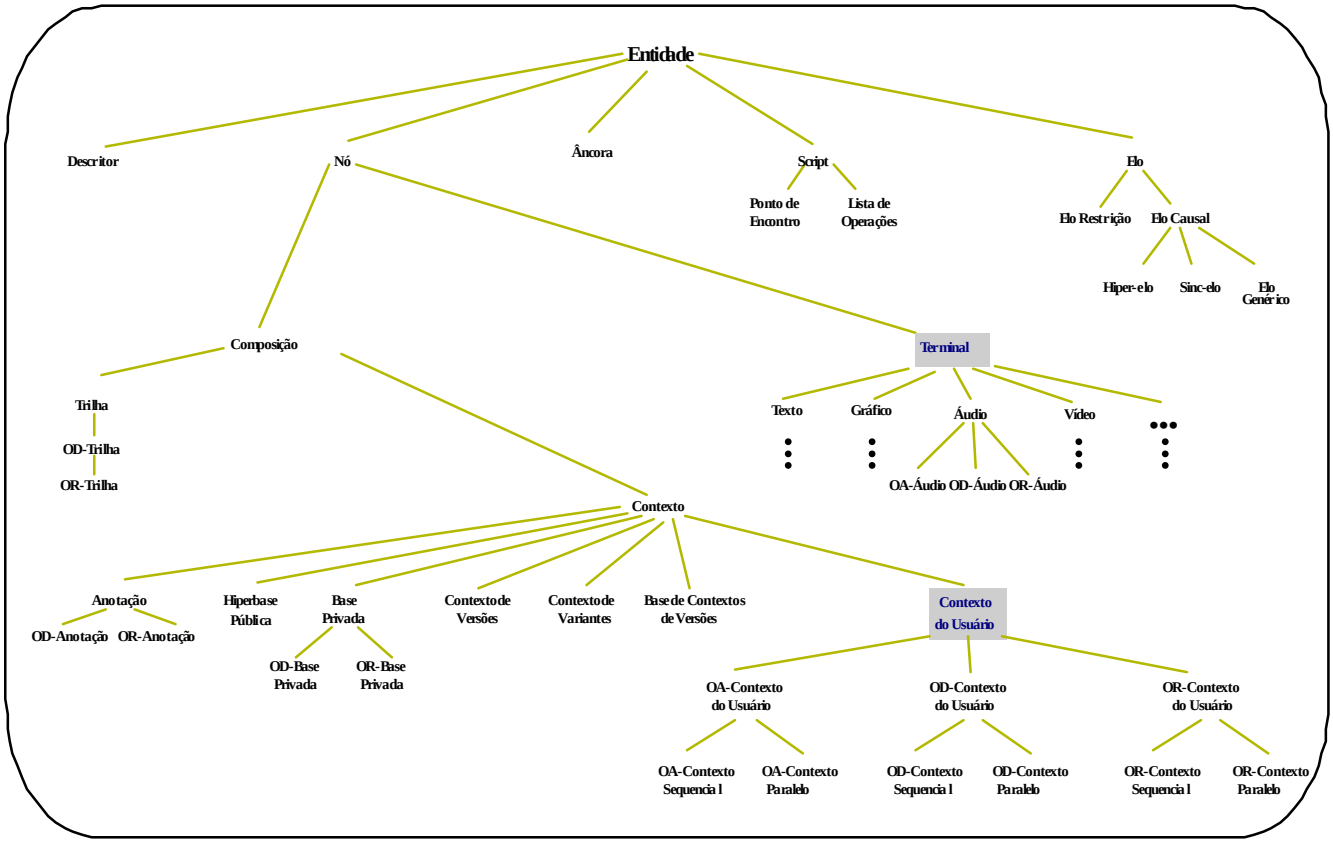


Figura 1: Hierarquia de Classes do NCM

Uma *entidade* possibilita que pares atributo/valor sejam associados a qualquer objeto. Cada entidade possui um identificador único (UID), hora e data de criação, autor e uma lista de controle de acesso (ACL). Cada entrada nesta lista associa um usuário, ou grupo de usuários, aos seus direitos de acesso a cada atributo do objeto.

Um *descritor* é uma entidade que contém definições que determinam como um objeto deve ser exibido para o usuário. Dessa forma, o mesmo objeto pode ser exibido de várias maneiras diferentes. Por exemplo, uma cadeia de caracteres pode ser exibida

como um texto utilizando um descritor D_{e1} ou pode ser sintetizada como um áudio usando outro descritor D_{e2}.

Um *nó* é uma entidade que possui como atributos adicionais um *conteúdo* e um *lista ordenada de âncoras*. Cada elemento da *lista ordenada de âncoras* é uma entidade âncora, chamada âncora do nó.

Uma *âncora* define um segmento, denominado *região*, dentro do conteúdo de um nó, agindo como uma interface externa do nó, no sentido de que qualquer entidade pode ter acesso a segmentos do conteúdo de um nó apenas através de seu conjunto de âncoras.

Um *nó terminal (ou de conteúdo)* é um nó cujo conteúdo e conjunto de âncoras é dependente da aplicação. O modelo permite que a classe de nós terminais seja especializada em outras classes (texto, áudio, vídeo, etc.). O conteúdo é formado por unidades de informação e a noção exata do que constitui uma unidade de informação é parte da definição do nó terminal. Por exemplo, uma unidade de informação de um nó vídeo pode ser um quadro, enquanto uma unidade de informação de um nó texto pode ser um caractere, ou uma palavra.

Um *nó de composição* é um nó cujo conteúdo é uma lista *L* de nós (de conteúdo e de composição, recursivamente) que se constituem em suas unidades de informação. Note que a região de uma âncora de um nó de composição *C* é uma sub-lista de *L*, ou toda a lista *L*.

Diz-se que uma entidade *E* em *L* é um *componente de C* e que *E está contido em C*. Como *L* é uma lista, uma mesma entidade pode estar contida mais de uma vez em *L*. Diz-se também que um nó *A* está *recursivamente contido em C* se e somente se *A* está contido em *C* ou *A* está contido em um nó recursivamente contido em *C*. Uma restrição importante é feita: um nó não pode estar recursivamente contido em si mesmo.

Nós de composição diferentes podem conter um mesmo nó e nós de composição podem ser aninhados em qualquer profundidade, desde que a restrição de um nó não conter recursivamente a si mesmo seja obedecida. Para identificar por qual seqüência de nós de composição aninhados uma dada instância de um nó *N* está sendo

observada, é introduzida a noção de perspectiva de um nó. A *perspectiva* de um nó *N* é uma seqüência P = (N_m,....,N_1), com $m \geq 1$, tal que $N_1 = N$, N_{i+1} é um nó de composição, N_i está contido em N_{i+1}, para $i \in [1,m)$ e N_m não está contido em qualquer nó. Note que pode haver várias perspectivas diferentes para um mesmo nó *N*, se este nó estiver contido em mais de uma composição. Dada a perspectiva P = (N_m,....,N_1), o nó N_1 é chamado *nó base da perspectiva*.

Um *nó de contexto T* é um nó de composição tal que seu conteúdo contém apenas nós terminais e nós de contexto, **sem repetição de componentes**. *T* pode ser especializado em outros tipos de nós, um dos quais o nó de contexto de usuário.

Um *nó de contexto de usuário U* é um nó de contexto cujo conteúdo é um conjunto *S* de nós terminais e nós contextos de usuário. *U* possui um atributo adicional, denominado *relações*, cujo conteúdo é um conjunto *R* de elos, tais que todo nó cabeça de cada elo em *R* ou é o próprio contexto de usuário *U*, ou um nó de *S* (a definição de elo e nó cabeça de um elo é dada logo abaixo).

Análogo aos nós de uma composição, diz-se que um elo *E* em *R* é um *componente de U* e que *E está contido em U*. Diz-se também que um elo *E* está *recursivamente contido em U* se e somente se *E* está contido em *U* ou *E* está contido em um nó recursivamente contido em *U*.

Um *elo E*, pertencente à um nó de composição *C*, determina uma relação *M*:*N* entre nós, e possui um conjunto de pontos terminais de origem e um conjunto de pontos terminais de destino. Um ponto terminal (origem ou destino) identifica uma âncora α de um nó N_1 em um dado aninhamento (N_k,..., N_2, N_1), onde N_{i+1} é um nó de contexto e N_i está contido em N_{i+1}, para todo $i \in (0,k)$, com k>0. O nó N_k, denominado *nó cabeça* do elo, ou está (diretamente) contido em *C*, ou é o próprio nó de composição *C*. O nó N_1 é chamado de *nó âncora* ou *nó base* do elo.

O conjunto de elos que ancoram ou passam por um nó, em uma dada perspectiva, é denominado de conjunto de *elos visíveis*.

Diz-se que um nó *A* referencia um nó *B* por elo quando *B* é o destino de um elo cuja origem é *A*. Diz-se também que um nó de composição *C* referencia um nó *D* por inclusão quando *C* contém *D* (diretamente).

2.2.1 Objetos de Armazenamento, Dados e de Representação

Um *objeto de dados* (*OD*) é criado ou como um objeto totalmente novo, ou como uma versão de um objeto persistente[1] já criado, denominado de objeto de armazenamento (*OA*).

A informação necessária para apresentar os componentes de um documento multimídia é fornecida pelo *objeto de representação* (*OR*), formado pela associação de um objeto de dados e um objeto da classe descritor. Um *descritor* especifica como um objeto de dados será exibido para o usuário, detalhando como será iniciado, qual dispositivo de E/S será utilizado e quais mudanças ocorrerão no seu comportamento durante a exibição, caso elas existam.

Em uma implementação cliente-servidor, é natural que os objetos de armazenamento fiquem no servidor, e os objetos de dados e de representação no cliente.

O modelo de apresentação possibilita a combinação de uma entidade objeto de dados com diferentes descritores, originando diferentes representações (objetos de representação) da mesma entidade. Suponha um objeto de representação OR_1 derivado a partir de um objeto de dados *OD* com um descritor D_1, e outro objeto de representação OR_2 derivado a partir do mesmo objeto de dados *OD* com um descritor diferente D_2. Alterações realizadas em OR_1 e OR_2 devem ser tratadas independentemente, logo, objetos de representação devem ser tratados como versões de representação.

---

[1] Uma informação em um programa é dita persistente quando pode ser recuperada após o término desse programa [POET 2000]. Logo, um objeto hipermídia é dito persistente quando, ao ser criado por uma aplicação, pode ser recuperado após o encerramento dessa aplicação.

A associação entre objetos de dados e descritores é representada na Figura 2 por linhas conectando os objetos de dados no plano intermediário com os objetos de representação, desenhados no plano superior. Na figura, nós são representados por círculos, elos por arcos e composições pela inclusão de círculos e arcos em círculos maiores, convenção que será adotada figuras dos Capítulos 2 e 3 desta dissertação.

Essa figura mostra a associação dos descritores D_{e1}, D_{e2} e D_{e3} ao objeto de dados *A'*, criando os objetos de representação $A"_1$, $A"_2$ e $A"_3$. O nó *A'* possui três diferentes representações porque existem, por exemplo, três diferentes formas de navegação até ele, por meio dos dois elos ou por meio da hierarquia de composições. Note assim que, devido ao fato de um mesmo objeto de dados poder gerar vários objetos de representação, um objeto de representação composição pode conter um número de elementos diferente da composição objeto de dados, por exemplo, o contexto de usuário *C"* da figura possui três nós, ao passo que o nó objeto de dados correspondente só possui um.

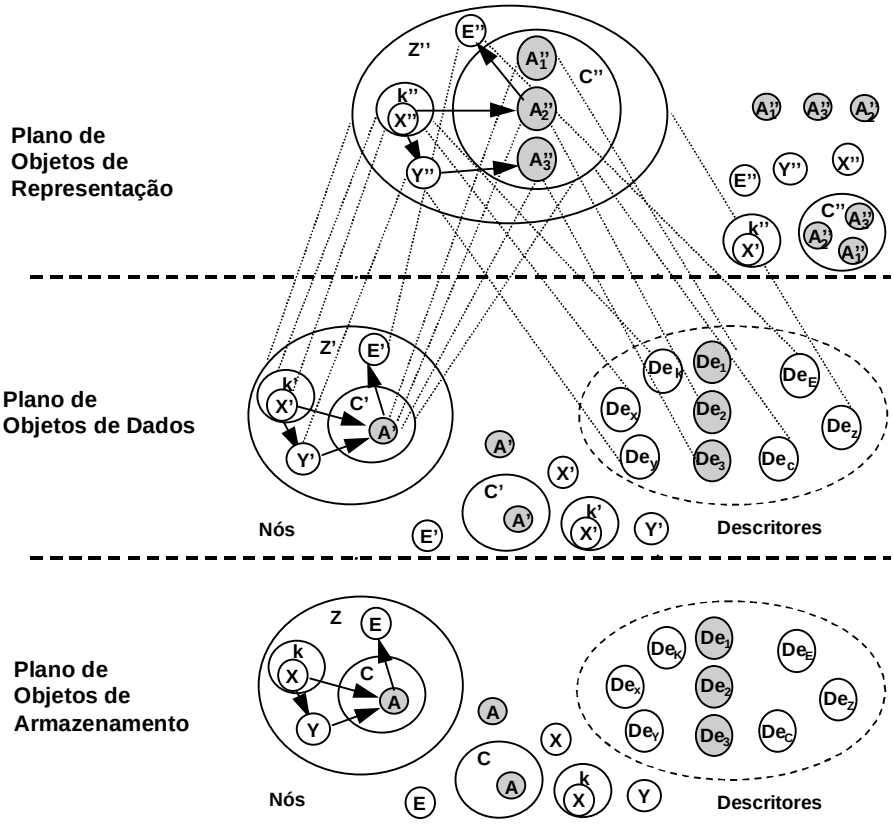


Figura 2: Planos de Objetos de Dados e de Representação

2.3 Controle de Versões no NCM

A referência [SOARES 1995] especifica alguns requisitos para um sistema de controle de versões, entre eles:

⇒ "R13. Representações Distintas da Mesma Informação — é importante que o sistema tenha como capturar a noção de que diferentes representações, em particular, em diferentes mídias, podem ser usadas para descrever uma mesma informação (como, por exemplo, a versão escrita e falada de uma palestra). Representações diferentes de uma mesma informação devem ser tratadas como diferentes versões desta informação.

⇒ R14. Uso Concorrente da Mesma Informação — cópias (temporárias) da mesma informação em uso em determinado instante devem ser consideradas versões desta informação. O acoplamento desta noção de versão com o mecanismo de notificação vai proporcionar um bom suporte a trabalhos cooperativos."

As versões de representação darão suporte ao atendimento do requisito 13, enquanto que as versões de dados darão suporte principalmente ao atendimento do requisito 14.

No NCM, apenas os nós terminais e nós de contexto de usuário (e suas subclasses) estão sujeitos a versionamento de dados, como salientado pelo fundo cinza na Figura 1. Além disso, cada atributo (incluindo o conteúdo) de um nó terminal ou contexto de usuário pode ser especificado como versionável ou não. Um atributo não versionável pode ter seu valor modificado sem a necessidade da criação de uma nova versão. Ao contrário, modificações no valor de um atributo versionável devem ser realizadas em uma nova versão do objeto, se este já estiver no estado permanente, como será detalhado na Seção 2.3.1.

A possibilidade de se adicionar novos atributos a um nó sem a criação de novas versões é bastante atraente. Suponha, por exemplo, que após um nó ser criado, uma nova ferramenta é introduzida no sistema. A ferramenta pode querer guardar algumas informações específicas nos nós. Em sintonia com esta possibilidade, no NCM o

usuário pode especificar se a adição de novos atributos é permitida sem a criação de novas versões do objeto.

Ao se criar uma versão de um nó, todos os elos, nos nós de contextos que recursivamente contêm a versão criada, devem ser apropriadamente atualizados de forma a refletir nos seus pontos terminais o novo nó criado e as novas âncoras criadas deste nó.

Versionamento de elos ainda não foi incluído no modelo, uma vez que acreditava-se que esta facilidade adicionaria mais complexidade do que funcionalidade ao sistema e, além disso, se necessário, poderia ser modelado pelo versionamento dos nós de contexto de usuário. Contudo, um estudo comparativo entre o modelo NCM e algumas linguagens para descrição de arquitetura [MUCHALUAT 2000] vem sendo realizado e como resultado desse estudo, novos tipos de elos poderão ser acrescentados ao modelo. Com isso, versionamento de elos tornará-se necessário e deverá ser incluído em trabalhos futuros.

2.3.1 Estado

Para facilitar o controle de consistência entre nós e a criação automática de versões, o NCM inclui a noção de estado em um nó terminal ou de contexto de usuário.

O *estado* é um novo atributo das classes nó terminal e nó de contexto de usuário que controla a execução de certas operações, e pode assumir os valores: *temporário*, *permanente* e *obsoleto*. Essas classes contam ainda com o método *altera estado*, que permite a alteração do valor do atributo *estado* de forma consistente. Esse método pode ser invocado explicitamente a partir de comandos do usuário ou implicitamente como efeito colateral de operações do modelo[2]. Por simplicidade, denomina-se *nó temporário* um nó em que o valor do atributo estado é igual a temporário, *nó permanente* um nó em que o valor do atributo estado é igual a permanente, e *nó obsoleto* um nó em que o valor do atributo estado é igual a obsoleto.

[2] Ao sofrer uma operação de derivação de versão, um nó terminal ou de contexto de usuário no estado temporário passa automaticamente para o estado permanente.

Um nó é criado no estado temporário e continua nesse estado enquanto seus atributos versionáveis estão sendo modificados. Quando alcançar um estado estável de edição, o nó pode ser promovido ao estado permanente. Um nó permanente não pode ser destruído diretamente ou sofrer alterações em seus atributos versionáveis, no entanto, o usuário pode derivar versões ou torná-lo obsoleto, permitindo que outros nós que o referenciem (por elo ou inclusão), ou dele derivados, sejam alertados. Um nó obsoleto é removido automaticamente por um processo de coleta de lixo do sistema, quando sua destruição não provoca inconsistências no hiperdocumento [Seção 2.3.1.1].

O conceito de estado de um nó é válido para todos os nós terminais ou de contexto de usuário, não importando a combinação de atributos versionáveis e não versionáveis que possuem. A Tabela 1 ilustra o comportamento de um nó terminal ou de contexto de usuário, dado seu estado.

Estado	**Atributos Modificáveis**	**Pode Derivar**	**Destruição**	**Mudança de Estado**	**Adição de Novos Atributos**	**Nós Internos***
Temporário	Todos	Versões de Representação	Pode	Permanente	Pode	Qualquer estado
Permanente	Os Não Versionáveis	Versões de Dados e de Representação	Não pode	Obsoleto	Possibilidade determinada pela aplicação	Permanentes ou obsoletos
Obsoleto	Nenhum	Não	Automática	Não pode	Não pode	Permanentes ou obsoletos

* Válido para nós de contexto de usuário

Tabela 1: Estados de um nó e possibilidade de operações

As operações de mudança de estado de um nó *N* (terminal ou de contexto de usuário) são sucedidas por mensagens de notificação, nas quais *N* envia uma notificação indicando que alcançou um estado estável de edição, tornou-se obsoleto ou foi destruído pelo sistema, como será visto no Capítulo 3.

2.3.1.1 Coleta de Lixo

A coleta de lixo é um processo de destruição de nós obsoletos, que é executado eventualmente. Esse processo remove nós em desuso de maneira a não provocar inconsistências no hiperdocumento, como por exemplo, elos ancorando em nós inexistentes.

Durante o processo de coleta de lixo, um nó obsoleto é mantido apenas enquanto está contido em outro nó de contexto de usuário ou deriva algum nó em uma base privada. Repare que a primeira regra evita com que um nó obsoleto *O*, referenciado por elo, seja removido. Isto ocorre porque, nesse caso, *O* está necessariamente contido em outro contexto [Seção 2.2], exceto quando *O* for um nó de contexto de usuário, referenciado por elo por um nó recursivamente contido em si, como mostra o exemplo da Figura 3. No último caso, o elo sempre pertence a *O* e pode ser removido normalmente já que o próprio *O* será destruído, não gerando qualquer inconsistência.

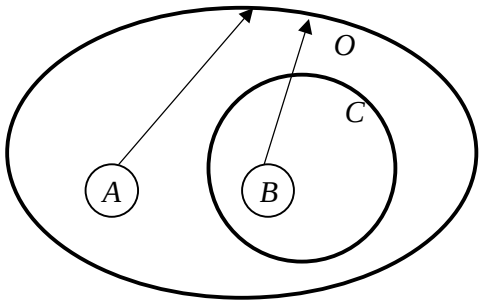


Figura 3: Nó de contexto de usuário *O* referenciado por elos internos

Quando um nó obsoleto *O* é destruído, notificações são disparadas, como será visto no Capítulo 3, e o contexto de versões onde *O* está inserido deve ser atualizado de forma que os nós derivados de *O* passem a ser considerados como derivados dos nós de onde *O* foi derivado. Caso *O* seja a raiz de derivação, os nós dele derivados passam agora a ser novas raízes. Observe também que na remoção de *O* pela coleta de lixo, os nós recursivamente contidos em *O* permanecem intactos.

2.3.2 Hiperbase Pública e Bases Privadas

De forma geral, um ambiente de autoria cooperativa deve permitir que usuários compartilhem informação e oferecer alguma maneira de criar informações privadas. A hiperbase pública corresponde a um repositório compartilhado de informação, formado por objetos de armazenamento. Uma base privada é utilizada para criação, edição e visualização de dados particulares de usuários, que podem ser objetos de dados ou de representação.

Uma *hiperbase pública* H_B é um nó de contexto que agrupa nós terminais e nós de contexto de usuário. Todos os nós em H_B devem ser nós objetos de armazenamento e podem estar ou no estado permanente ou no obsoleto. Como em toda hiperbase, se um nó de contexto de usuário *C* está em H_B, então todos os nós contidos em *C* devem também estar contidos em H_B.

Uma *base privada BP* é também uma especialização do nó de contexto, cujo conteúdo é formado por nós de contexto de usuário, nós terminais, nós de anotação[3] e bases privadas. *BP* possui um atributo adicional, denominado *relações*, cujo conteúdo é um conjunto *R* de elos. Os elos em *BP* sempre tem como nó cabeça de seu ponto terminal de origem um nó de anotação. A classe base privada é especializada em OD-base privada e OR-base privada.

Uma *OD-base privada ODBP* é um objeto de dados que só contém objetos de dados, e se um nó de composição *N* está contido na *ODBP*, seus componentes ou estão contidos em *ODBP*, ou na hiperbase pública, ou em qualquer base privada de um aninhamento de bases privadas onde *ODBP* está recursivamente contida.

Uma OR-*base privada ORBP* é um objeto de representação que só contém objetos de representação, e se um nó de composição *N* está contido na *ORBP*, seus componentes ou estão contidos em *ORBP*, ou em qualquer base privada de um aninhamento de bases privadas onde *ORBP* está recursivamente contida, ou contidos

[3] Anotações consistem de comentários (em qualquer formato ou mídia) e mantêm referências às versões, objeto do comentário, e às versões, consideradas respostas ao comentário.

na OD-base privada *ODBP* de onde *ORBP* foi derivada[4], ou em qualquer base privada de um aninhamento de bases privadas onde *ODBP* está recursivamente contida, ou então contidos na hiperbase pública.

Uma versão específica de um nó terminal ou de contexto de usuário pode pertencer a apenas uma base, seja ela uma *ORBP*, uma *ODBP* ou a hiperbase pública.

Seja *P*" uma perspectiva para o nó *N*, objeto de dados ou de armazenamento, em uma OR-base privada *ORBP*. Seja *P'*, a perspectiva correspondente a *P*" na OD-base privada correspondente a *ORBP*, contendo *N* como nó base e substituindo todos os nós de representação de *P*" pelos nós de dados de onde foram derivados. P" é denominada de *perspectiva correlata* de P' e vice-versa.

2.3.3 Histórico de Derivação

O controle de versões deve prover meios para que o histórico de derivação de um determinado nó seja registrado. Esse histórico deve ser capaz de representar diferentes formas de criação de versões de um nó, como derivações 1:N e N:1. A derivação 1:N ocorre quando um determinado nó gera várias versões, e a derivação N:1 ocorre quando vários nós geram uma única versão. No modelo WebDAV [WEBDAV 2000], essas derivações são chamadas respectivamente de ramificação (*branch*) e fusão (*merge*)[5].

No NCM, os nós *Contexto de Versões*, *Base de Contexto de Versões* e *Contexto de Variantes*, os quais serão definidos a seguir, são responsáveis pela manutenção do histórico de derivação de um documento.

---

[4] Seja *ODBP* uma OD-base privada que deriva uma OR-base privada *ORBP*. É dito que *ODBP* é correspondente a *ORBP*.

[5] No decorrer desta dissertação, o termo "ramificação" será usado para representar derivações 1:*N*. O termo "fusão" será utilizado para representar derivações *N*:1, não importando como e em que momento tais derivações aconteceram (por exemplo, no *check-out*, no *check-in*, primitivas que serão definidas adiante).

2.3.3.1 Contexto de Versões

Um *nó de contexto de versões CV* é um nó de contexto cujo conteúdo é um conjunto *S* formado ou por nós terminais, ou por nós contextos de usuário[6]. *CV* agrupa nós objetos de armazenamento ou objetos de dados que representam versões de dados de uma mesma entidade[7]. Todos os nós pertencentes à *S* são chamados de *versões correlatas* e não precisam pertencer a mesma classe (texto, áudio, vídeo, etc.).

CV possui um atributo adicional, denominado *relações*, cujo conteúdo é um conjunto *R* de elos da forma (<v_1, i_1, ϕ, ϕ>, <v_2, i_2, ϕ, ϕ>), indicando que v_2 é derivada de v_1, onde v_1 e v_2 estão contidas em *CV*. As âncoras de um elo em *CV* servem para indicar que partes de v_1 geraram quais partes de v_2, no caso de versionamento devido a alterações do atributo conteúdo do nó. Note que um elo em um nó de contexto de versões relaciona somente duas versões, logo, para expressar as operações de ramificação e fusão de nós, vários elos são necessários.

A Figura 4 exibe um exemplo de um nó de contexto de versões que agrupa as versões V_1 , V_2, V_3 e V_4, onde V_2 é derivada de V_1, V_3 é derivada de V_2, e V_4 também derivada de V_2.

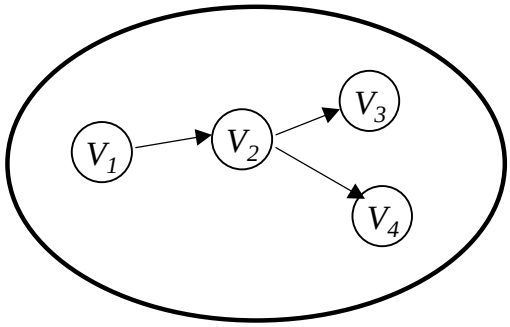


Figura 4: Exemplo de um nó de contexto de versões

[6] De agora em diante, a seguinte convenção será adotada: quando for utilizado como exemplo um nó *V*, este será necessariamente ou um nó terminal ou um nó de contexto de usuário.

[7] Note que versões de representação não são incluídas nos contextos de versões.

Por simplicidade, um nó *N* que pertence ao conjunto *S* de um nó de contexto de versões CV_1 é dito como pertencente à CV_1, e um elo *E* que pertence ao conjunto *R* de um nó CV_1 é dito também como pertencente à CV_1.

Nós e elos em um contexto de versões devem ser inseridos e removidos automaticamente pelo sistema, na ocorrência de operações de versionamento, que serão vistas na Seção 2.3.4. Por exemplo, quando um nó *N* deriva um nó *N'*, supondo *CV* o contexto de versões de *N*, tem-se:

1) *N'* é inserido em *CV* e

2) Um elo de *N* para *N'*, representando a derivação, é inserido em *CV*.

Para indicar explicitamente que determinados nós são versões do mesmo objeto, um usuário poderá manualmente adicioná-los a um contexto de versões, e para indicar como as versões foram derivadas, o usuário poderá inserir novos elos. Contudo, para a preservação da consistência, a criação manual de elos de derivação causa, automaticamente, a mudança do nó origem *N* do elo para o estado permanente, se *N* estiver no estado temporário.

Os elos em um contexto de versões não estão sujeitos a restrições, exceto que o grafo induzido por eles deve ser acíclico.

Uma aplicação poderá definir, de várias formas, um nó em um contexto de versões *V* como a sua *versão corrente*, escolhida de acordo com um critério específico. No caso mais simples, a aplicação poderá reservar uma âncora de *V* para manter uma referência à sua versão corrente *V*. Outras âncoras poderão especificar outras versões de acordo com outros critérios de seleção. No modelo com entidades virtuais, a aplicação poderá se valer de consultas para localizar dinamicamente a sua versão corrente. A consulta poderá ser armazenada em uma âncora ou definida em um elo fora do contexto de versões.

A seleção de versão corrente baseada em uma linguagem de consulta é uma técnica poderosa, mas pode aumentar o *overhead* cognitivo do usuário, que tem de definir o critério de seleção para cada elo criado. Para atenuar esse problema, no NCM cada nó de contexto de versões tem duas âncoras especiais, definidas por consultas

default, para a recuperação da versão corrente. Uma destas consultas *default* é definida no próprio contexto de versões. A outra é especificada, de uma forma mais geral, em um atributo da base privada. Quando um elo é criado, o nó de destino é examinado. Se o nó é um componente de um nó de contexto de versões não especificado explicitamente pelo usuário (diretamente ou através de uma consulta), o elo é criado usando o critério de seleção especificado na base privada que contém o nó de contexto onde o elo está contido. Se a consulta não estiver especificada nesta base privada, o elo usará a consulta *default* definida no contexto de versões.

2.3.3.2 Base de Contexto de Versões

Um *nó base de contextos de versões BV* é um nó de contexto cujo conteúdo é um conjunto *S* formado por nós de contexto de versões. Por definição, todo nó de contexto de versões está contido em uma única base de contexto de versões.

BV possui um atributo adicional, denominado *relações*, cujo conteúdo é um conjunto *R* de elos da forma (<C_1, v_1, i_1, ϕ, ϕ>, < C_2, v_2, i_2, ϕ, ϕ>), indicando que v_2, contida no contexto de versões C_2, é derivada de v_1, contida no contexto de versões C_1, onde C_1 e C_2 estão contidos em *BV*. Como para os contextos de versões, as âncoras de um elo em *BV* servem para indicar quais partes de v_1 geraram quais partes de v_2, e um elo pertencente à base de contexto de versões relaciona também somente duas versões.

A Figura 5 exibe um exemplo de um nó base de contexto de versões que agrupa os contextos de versões de *A*, *B* e *C*. Note a existência de uma fusão dos nós A_3 e *B*, pertencentes à contextos de versões diferentes.

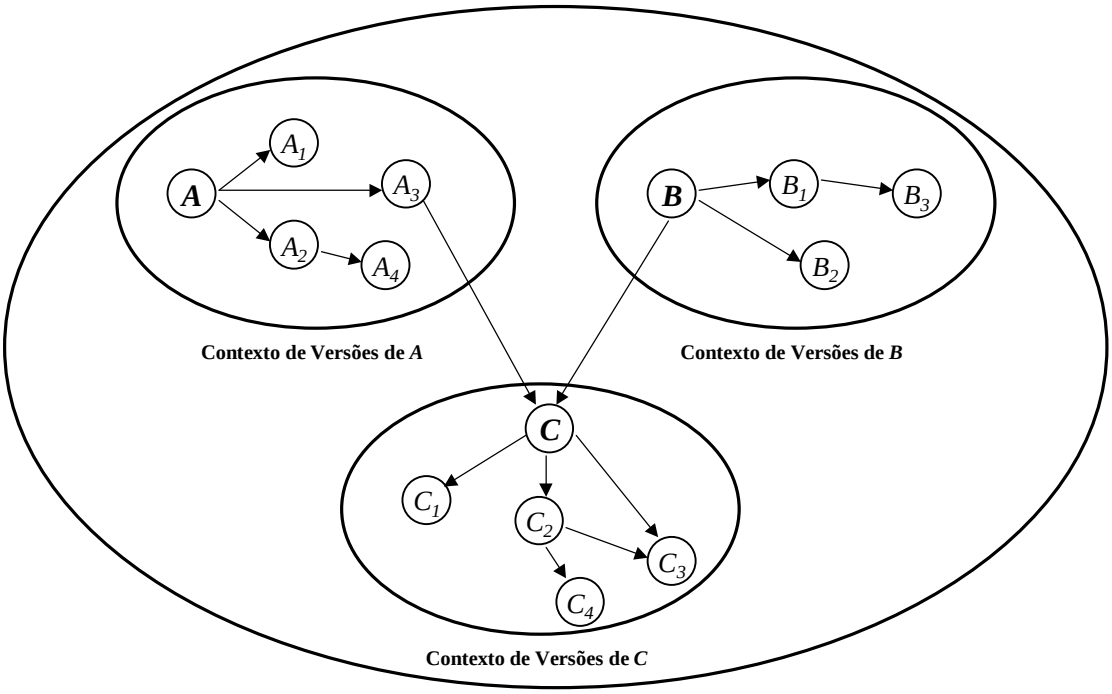


Figura 5: Exemplo de um nó base de contexto de versões

Novamente por simplicidade, um nó de contexto de versões *CV* que pertence ao conjunto *S* de um nó base de contexto de versões BV_1 é dito como pertencente à BV_1, e um elo *E* que pertence ao conjunto *R* de um nó base de contexto de versões BV_1 é dito também como pertencente à BV_1.

Para indicar como as versões foram derivadas, o usuário poderá também inserir novos elos na base de contexto de versões, desde que esses elos não tornem o grafo de versões cíclico. Como no contexto de versões, para a preservação da consistência, a criação manual de elos de derivação causa, automaticamente, a mudança do nó origem *N* do elo para o estado permanente, se *N* estiver no estado temporário.

No NCM, a hiperbase pública tem uma base de contextos de versões associada, de forma que todas as versões de dados derivadas de seus nós de armazenamento são inseridas automaticamente nos contextos de versões que contêm este nós de armazenamento, contextos de versões que por sua vez estão contidos na base de contextos de versões associada.

A razão da estruturação do grafo de versões de armazenamento e de dados em contextos de versões contidos em uma base de contextos de versões segue o seguinte raciocínio. Para delimitar o escopo do grafo para a definição de versões correntes foi

criado o conceito de contextos de versões. Contudo, apenas os contextos de versões não seriam suficientes, pois como uma versão pode ser derivada de vários nós (fusão) que podem estar em contextos de versões diferentes, é necessária a definição da base de contextos de versões para capturar essas relações de derivação, que são representadas em seus elos.

A definição da base de contexto de versões como uma nova entidade no NCM para solucionar o problema da representação de derivações N:1, quando as versões a serem fundidas pertencem a contextos de versões diferentes, é uma das contribuições dessa dissertação. Na verdade, a operação de fusão de nós ainda não está definida no NCM, no entanto, já é possível representá-la no grafo de derivação.

Várias outras soluções possíveis foram consideradas, conforme descrito nos próximos parágrafos, mas todas possuíam pequenas desvantagens frente à solução proposta. Para a descrição das outras soluções, suponha que ocorreram as derivações representadas na Figura 6. Suponha também que seja necessário representar a fusão dos nós *A* e *B*, gerando uma nova versão *V*.

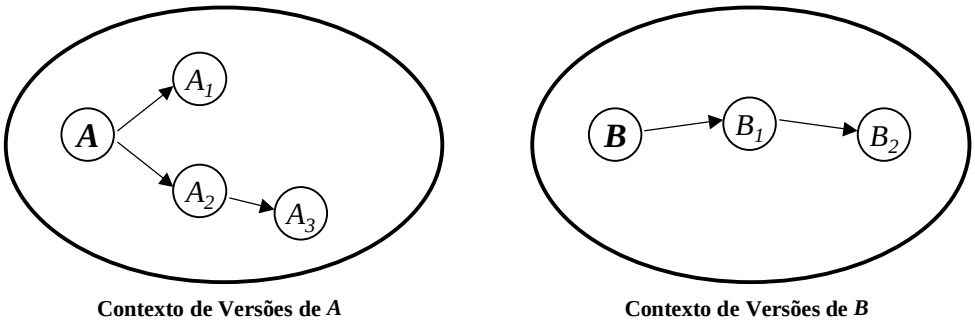


Figura 6: Suposição de derivações de versões dos nós *A* e *B*

Uma solução seria unir os contextos de versões de *A* e *B* em um contexto de versões *AB* e explicitar as derivações em *AB*, como mostra a Figura 7. O problema é que tanto as versões correlatas de *A* como as de *B* pertenceriam ao mesmo contexto de versões *AB*, fazendo com que *AB* deixasse de agrupar versões correlatas. As noções de versão corrente (e outras características definidas) de *A* e de *B* também seriam perdidas.

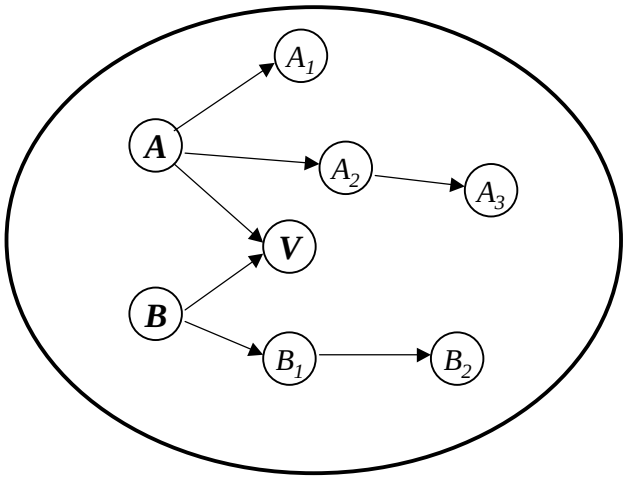


Figura 7: União dos contextos de versões de *A* e de *B*

Outra solução seria incluir *V* nos contextos de versões de *A* e de *B*, como derivado de *A* e *B*, respectivamente, e criar um novo contexto de versões para *V*, como mostra a Figura 8. Nesse caso, um mesmo nó poderia pertencer a vários contextos de versões, obrigando a existência de uma informação identificando de quais nós um determinado nó foi derivado.

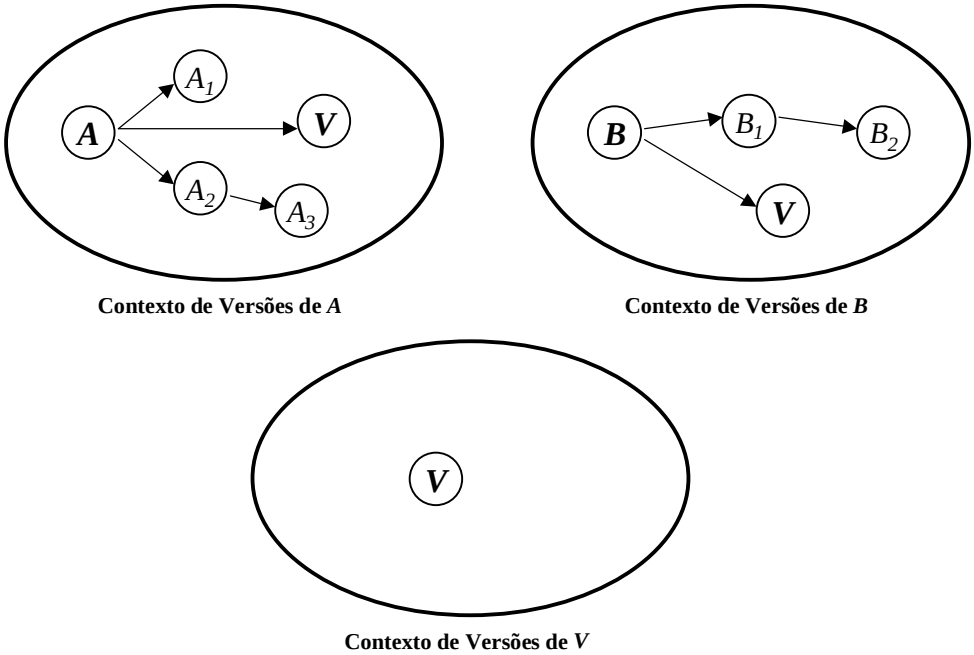


Figura 8: Inclusão de *V* em vários contextos de versões

Pode-se pensar ainda em um aninhamento de contextos de versões. Nesse caso, um contexto de versões de *V* seria criado envolvendo os contextos de versões de *A* e de *B*,

como mostra a Figura 9. Essa solução não é a mais simples e seria ainda mais complexa, caso uma nova fusão de *V* e *B*, por exemplo, fosse realizada. O novo contexto de versões conteria *V* (que contém os contextos de versões de *A* e de *B*) e também o de *B*, fazendo com que um contexto de versões pertencesse a vários contextos de versões.

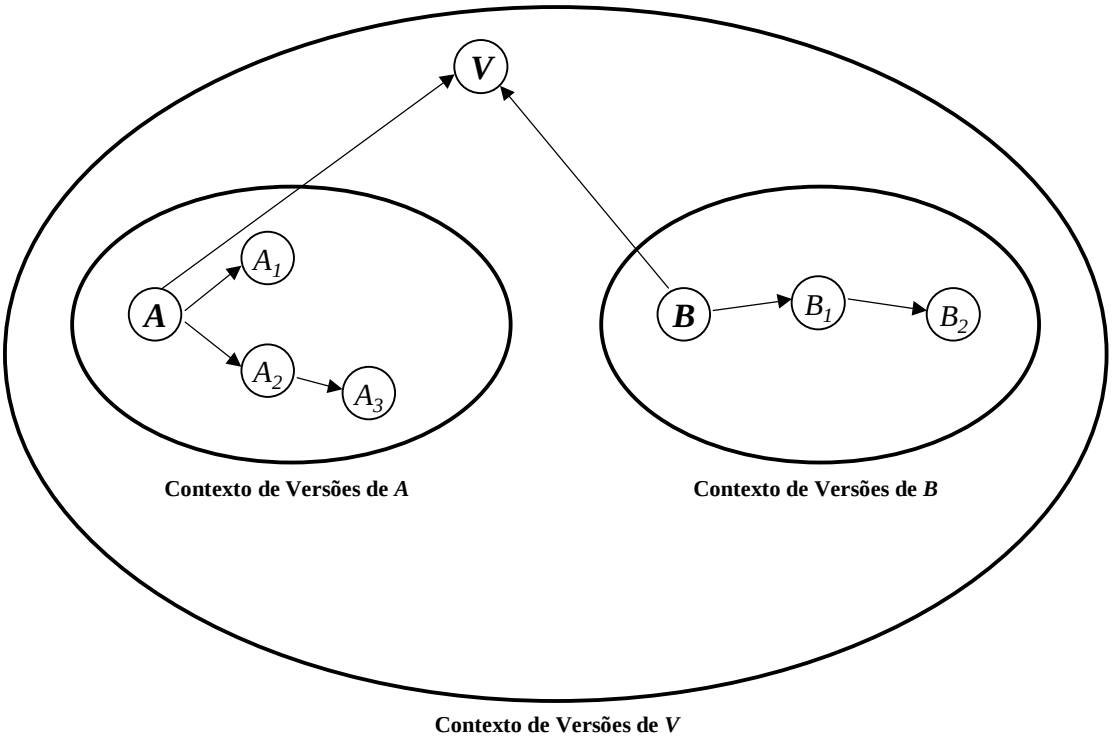


Figura 9: Aninhamento de contextos de versões

2.3.3.3 Contexto de Variantes

Um *nó de contexto de variantes V* é um nó de contexto cujo conteúdo é um conjunto *S* que contém ou nós terminais ou nós de contexto de usuário. Todos os nós de um contexto de variantes são objetos de representação que representam versões de representação do mesmo objeto de dados, logo, não há necessidade de elos para explicitar relações de derivação e, consequentemente, nem de uma base de contexto de variantes.

A Figura 10 exibe um exemplo de um nó de contexto de variantes composto pelas versões de representação V_1, V_2, V_3, V_4 e V_5.

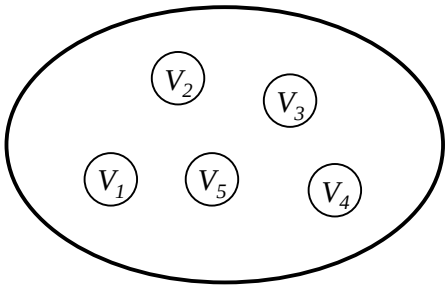


Figura 10: Exemplo de um nó de contexto de variantes

Também por simplicidade, um nó de contexto de variantes CT_1 que pertence ao conjunto S de um nó base de contexto de versões CT_1 é dito como pertencente à CT_1.

O NCM poderia ter usado os nós de contexto de versões para armazenar todos os tipos de versões (de dados e de representação) de um nó. Existem, no entanto, três razões pelas quais isto não foi feito:

A primeira é que versões de representação podem ser derivadas de nós no estado temporário, o que significa que a consistência de dados não é necessariamente mantida no contexto de variantes, em contraposição com o que acontece com os contextos de versões. Na realidade, tal consistência será mantida se as mudanças nos objetos de dados forem realizadas por primitivas de *check-in* (a ser posteriormente definida) entre os planos de objetos de representação e de objetos de dados, mas o NCM não tem tal restrição.

A segunda razão vem do fato que em uma implementação cliente-servidor, é natural que o servidor mantenha os contextos de versões. Contudo, operações de criação de versões de representação de um mesmo objeto de dados estão geralmente restritas a um único cliente. Dessa forma, é conveniente que os clientes mantenham a estrutura de derivação das versões de representação, contidas nos contextos de variante, evitando trocas desnecessárias de mensagens.

A terceira razão é que, em uma implementação, versões de representações não necessitam de persistência, elas existem somente em tempo de navegação ou edição. Consequentemente, os contextos de variantes também não precisam ser tornados persistentes, diferentemente dos contextos de versões.

2.3.3.4 Grafo de Versões

O *grafo de versões* (armazenamento, dados e representação) pode então ser pensado como a concatenação dos nós e elos pertencentes a seus componentes distribuídos (contextos de versões e de variantes), e dos elos definidos na base de contexto de versões correspondente. A consistência é garantida em todo grafo, exceto nos seus vértices sumidouros[8], definidos pelas versões de representação.

Suponha que existam dois arcos em um grafo de versões qualquer, o primeiro possuindo como origem um nó *V* e destino outro nó *V'*, e o segundo possuindo como origem *V* e destino outro nó *V''*. Ou seja, *V* deriva *V'* e *V''*. Denomina-se que:

⇒ *V* é *ascendente* no grafo de derivação de *V'* e *V''*

⇒ *V'* e *V"* são *descendentes* no grafo de derivação de *V*

⇒ *V'* e *V''* são *irmãos* no grafo de derivação

Seja *V* um nó qualquer, e *P* um de seus ascendentes. *P* é denominado também de *ancestral* de *V*. Um ascendente de um ancestral de *V* é também denominado de ancestral de *V*.

Essas definições serão importantes para propagação de mensagens de notificação, como será visto na Seção 3.3.

2.3.4 Tratamento de Versões em Bases de Documentos

Para ter acesso a um dado multimídia no NCM, uma aplicação identifica o objeto de armazenamento *A* contendo o dado desejado. Uma versão de dados *D* é então criada a partir de *A*, que pertence ao repositório público (hiperbase pública), e inserida em uma OD-base privada *ODBP*. Em seguida, uma versão de representação *R* é derivada a partir da associação de *D* a um objeto da classe descritor e inserida na OR-base

[8] Em um grafo direcionado *G*, um vértice sumidouro é um vértice pertencente ao conjunto de nós de *G*, cujo grau de saída é igual a zero, isto é, não possui arestas de saída [SZWARCFITER 1984].

privada correspondente a *ODBP*. Eventualmente, a versão de representação *R* poderá ser movida para *ODBP*, e a versão de dados *D*, para a hiperbase pública.

As primitivas que permitem tais operações serão descritas nas próximas seções.

2.3.4.1 Check-out

A primitiva *check-out (P, C_{ont})* permite a criação de uma nova **versão de dados** temporária de um nó *N* (objeto de dados ou de armazenamento) em uma OD-base privada *ODBP*, onde *N* (que deve estar no estado permanente) é o nó base da perspectiva *P*. Todos os nós de *P* devem ser nós objeto de dados, exceto o nó *N*. Se já existir uma versão de dados derivada de *N* em *ODBP*, recursivamente contida no contexto C_{ont}, a nova versão não é criada. C_{ont} pode receber o valor nulo (ϕ), obrigando que uma nova versão seja sempre criada. A versão temporária *D* quando criada é exatamente idêntica à *N*, ou seja, se *N* for um nó de contexto de usuário, *D* conterá os mesmos nós e elos de *N*. *D* deverá ser incluída em *ODBP*. Caso a versão *D* seja criada, ou se a versão *D* já existir[9], se *N* na perspectiva *P* estiver contido em um contexto *C* diferente de *ODBP*, então:

1. Se C está no estado temporário,

D é incluído em *C*. Se *N* é um nó objeto de armazenamento, *N* deve ser removido de *C*. Neste caso, todos os elos visíveis em *P* por *N*, se existirem, são alterados de forma a refletir apropriadamente, nos seus pontos terminais, o novo nó *D*, em substituição à *N*.

2. Se C está no estado permanente,

Deve-se percorrer a perspectiva $P = (ODBP,..., N_{k-1}, N_{k-1},...,C,N)$, em direção à OD-base privada, até que se encontre o primeiro contexto no estado temporário ou

[9] Caso haja mais de uma versão recursivamente contida em C_{ont} , a aplicação pode adotar o procedimento de escolha. Ela pode consultar o usuário ou seguir uma opção definida no sistema como "Sempre utilizar a primeira versão encontrada".

até que se chegue na base privada. Seja N_k o contexto encontrado na perspectiva *P*. É então criado na *ODBP*, e incluído em N_k (se diferente da *ODBP*), um contexto no estado temporário N^*_{k-1} idêntico ao contexto N_{k-1} da perspectiva. No contexto de versões apropriado, N^*_{k-1} é inserido como derivando de N_{k-1}. De forma análoga, é criado na *ODBP* o nó N^*_{k-2}, no estado temporário, idêntico a N_{k-2}. O nó N_{k-2}, contido em N^*_{k-1}, é substituído por N^*_{k-2}, e assim sucessivamente até o nó *C*, inclusive, que será substituído pelo nó *C**. Nos contextos de versões apropriados, N^*_{k-i} (I>2) é inserido como derivando de N_{k-i}. Todas as referências por composição ou elo recursivamente contidas em N_{k-1} devem ser modificadas para refletir os novos nós criados. Uma nova perspectiva *P** é construída a partir de *P*, substituindo os nós N_{k-i} pelos nós N^*_{k-i} correspondentes. O procedimento passa agora a ser idêntico ao caso 1, onde *C* é substituído por *C** e *P* por *P**.

Quando no evento *check-out* ocorre efetivamente a criação de uma versão *D*, notificações são disparadas e regras automáticas que geram inscrições (que recebem notificações) são acionadas, como será visto no Capítulo 3.

Uma operação de versionamento composta bastante útil, denominada operação *Open (P,N)* utiliza a operação *Check-out (P, N)*, onde *N* é um nó de contexto de usuário no estado permanente (objeto de armazenamento ou de dados) e nó base da perspectiva *P*.

Primeiramente, a primitiva *Open(P,N)* aciona a primitiva *Check-out (P, N)*, onde a versão criada ou existente, na execução dessa última primitiva, é chamada de *D*. Então, a primitiva *Check-out (P', N)* é executada em cada nó terminal permanente contido em *N*, seguida pela execução da primitiva *Open (P', N)* em cada contexto de usuário permanente contido em *N*, e assim recursivamente e nessa ordem, onde *P'* é formado pela perspectiva *P* (com *N* substituído por *D*, versão criada ou existente, no *Check-out (P, N)*) acrescida do nó, em versionamento, contido em *D*. A ordem de aplicação das primitivas é importante e deve seguida de forma a garantir um resultado único. *D* conterá, ao final do processo, as novas versões de dados dos componentes de *N*, e seus elos serão criados de forma a refletir apropriadamente os elos de *N*.

Algumas conseqüências interessantes decorrem do comportamento diferente da operação *open* e a primitiva *check-out* para a criação de versões de dados. De fato, seja *A* um nó da hiperbase pública e *N* um nó contido recursivamente em *A* através de duas perspectivas. Suponha inicialmente que a versão de dados *D* tenha sido criada de *A* pela primitiva *check-out*. Observe que, pela definição de *check-out*, temos que *A* e, consequentemente, *N* são necessariamente permanentes e que *D* conterá recursivamente *N* exatamente como *A* contém. Ao atualizar *N* através de uma das perspectivas, o usuário criará então uma versão de dados de *N*, que não será visível pela outra perspectiva. Portanto, ao tentar atualizar *N* pela outra perspectiva, o usuário criará uma segunda versão de dados. Suponha agora que *D* tenha sido criada através da operação *open*. Neste caso, *D* já conterá uma (única) versão de dados *N'* de *N*, temporária, que poderá ser atualizada por quaisquer das duas perspectivas.

Quando, na primitiva *check-out (P, C_{ont})*, o nó base *N* de *P* é um objeto de armazenamento, *N* deve ser substituído também na OR-base privada, derivada da OD-base privada onde *P* é definida, pela versão criada *D*, em todas as perspectivas *P'* correlatas de *P*. Todos os elos visíveis em *P'* por *N*, se existirem, devem ser alterados de forma a refletir apropriadamente, nos seus pontos terminais, o novo nó *D*, em substituição a *N*. Note que, no caso da primitiva *Open (P,N)*, estes elos visíveis vão então refletir apropriadamente, nos seus pontos terminais, todas as versões de dados criadas, recursivamente contidas em *D*, se *D* é um nó de contexto de usuário, em substituição aos nós dos quais as versões foram derivadas.

A primitiva *Check-out (P, D_e, C_{ont})* permite a criação de uma nova **versão de representação** de um objeto de dados *D*, onde *D* (no estado permanente ou temporário) é o nó base da perspectiva *P*. Todos os nós de *P* devem ser nós objeto de representação, exceto o nó *D*. A versão, que deve ser incluída na OR-base privada *ORBP*, é criada pela associação do descritor D_e especificado, ao nó *D*. Se já existir uma versão de representação de *D* em *ORBP*, derivada do mesmo descritor D_e, e recursivamente contida no contexto C_{ont}, a nova versão não é criada. C_{ont} pode receber o valor nulo (ϕ), obrigando que uma nova versão seja sempre criada. Como restrição, se *D* for um nó de contexto de usuário, *D* só pode ter uma versão de representação em

uma mesma perspectiva. O objetivo desta restrição é simplificar o tratamento de versões, principalmente na duplicação de elos.

Seja *R* a versão de representação do objeto de dados *D* ou criada em *ORBP* pela agregação do descritor D_e através da execução da operação *Check-out (P*, D_e, C_{ont}*)*, ou então já existente em *ORBP* quando da execução da operação[10]. Suponha que *D* na perspectiva *P* está contido em um contexto *C* diferente de *ORBP*, então:

1. Se C está no estado temporário e

   i) se não existirem outras referências ao nó *D* com outros descritores (referências por elo ou composição), especificadas nos vários nós que compõem a perspectiva *P*, *R* é incluído em *C* e *D* é removido de *C*. Todos os elos visíveis em *P* por *D*, se existirem, são alterados de forma a refletir apropriadamente, nos seus pontos terminais, o novo nó *R*, em substituição a *D*.

   ii) se existirem outras referências ao nó *D*, especificadas nos vários nós que compõem a perspectiva *P*, com outros descritores (referências por elo ou composição), *D* não é removido de *C*. *R* é incluído em *C* e todos os elos visíveis em *P* por *D* que têm *D* como nó âncora (de destino ou fonte) e D_e como descritor devem ser apropriadamente direcionados para *R*. Todos os elos visíveis em *P* por *D* que têm *D* como nó âncora e um descritor diferente de D_e devem permanecer inalterados.

2. Se C está no estado permanente,

Deve-se percorrer a perspectiva *P*= (*ORBP*,..., N_k, N_{k-1},...,*C*,*D*), em direção à OR-base privada, até que se encontre o primeiro contexto no estado temporário ou até que se chegue na base privada. Seja N_k o contexto encontrado na perspectiva *P*. É então criado na *ORBP*, e incluído em N_k (se diferente da *ORBP*), um contexto no estado temporário N^*_{k-1} idêntico ao contexto N_{k-1} da perspectiva. No contexto de variantes apropriado, N^*_{k-1} é inserido como derivando do mesmo nó de onde derivou

[10] Como em versões de dados, caso haja mais de uma versão recursivamente contida em C_{ont} , a aplicação pode adotar o procedimento de escolha. Ela pode consultar o usuário ou seguir uma opção definida no sistema como "Sempre utilizar a primeira versão encontrada".

N_{k-1}. De forma análoga, é criado na *ORBP* o nó N^*_{k-2}, no estado temporário, idêntico a N_{k-2}. O nó N_{k-2}, contido em N^*_{k-1}, é substituído por N^*_{k-2}, e assim sucessivamente até o nó *C*, inclusive, que será substituído pelo nó *C**. Nos contextos de variantes apropriados, N^*_{k-i} (I>2) é inserido como derivando do mesmo nó de onde derivou N_{k-i}. Todas as referências por composição ou elo recursivamente contidas em N_{k-1} devem ser modificadas para refletir os novos nós criados. Uma nova perspectiva *P** é construída a partir de *P*, substituindo os nós N_{k-i} pelos nós N^*_{k-i} correspondentes. O procedimento passa agora a ser idêntico ao caso 1, onde *C* é substituído por *C** e *P* por *P**.

2.3.4.2 Efeitos da Mudança de Estado em Versões de Representação

Antes de se passar às primitivas de *check-in*, é necessário especificar o que acontece com um nó na OD-base privada quando uma de suas versões na OR-base privada é tornada permanente. Esta discussão é necessária, pois permitiu-se o versionamento de representação de um nó de dados no estado permanente ou temporário.

Suponha que um objeto de dados *D*, nó base de uma ou mais perspectivas P_i, seja a versão de um objeto de armazenamento *A*, ou um objeto de dados novo. Seja C_i o objeto de dados que contém *D* em P_i. Seja ainda C'_i um nó de contexto, dito relacionado a C_i, que contém uma ou mais versões de representação de *D*, tal que C'_i é uma OR-base privada, no caso em que C_i é uma OD-base privada, ou então uma versão de representação de C_i, em caso contrário. Quando mais de um objeto de representação R_1,.., R_j,.. R_n for derivado de *D*, se um objeto de representação R_j qualquer passar do estado temporário para permanente, diversas alternativas podem ocorrer:

1) Qualquer atributo versionável de R_j sofreu alterações e *D* está no estado temporário. Neste caso, é criado um novo objeto de dados D_j correspondente a R_j. D_j estará no estado permanente, terá como conjunto de descritores alternativos o mesmo conjunto de descritores alternativos de *D*, e será incluído na OD-base privada e no contexto de versões correspondente como derivado de *A*, caso não seja um objeto

novo. R_j é removido do contexto de variantes como derivado de D e inserido no novo contexto de variantes como derivado de D_j. D_j será incluído em todas as composições C_is relacionadas a C'_is tais que R_j está contido em C'_i. Nessas composições C_is, se R_j é o nó de representação criado na navegação em profundidade quando se seleciona D, o conjunto de descritores alternativos associados ao nó D em C_i para criação de R_j deve ser removido e agora fazer parte da lista de conjunto de descritores de D_j. Nas composições C_is, os elos visíveis por D em P_i, que tinham D como nó âncora (de destino ou fonte) e o descritor usado para criar R_j, devem agora ser apropriadamente direcionados para D_j. Todos os elos visíveis por D em P_i que têm D como âncora e um descritor diferente do descritor usado para criar R_j devem permanecer inalterados. Todos os demais elos visíveis por D em P_i que têm D na lista de nós de um ponto terminal devem ter D substituído por D_j em seus pontos terminais. O usuário será notificado da criação de um novo objeto de dados, por meio da versão D, como será visto no Capítulo 3. Se R_j é o último nó de representação que passa para o estado permanente em C'_i e se o nó D não possui mais qualquer referência por elo e nem descritor especificado em C_i, D é removido de C_i.

2) Qualquer atributo versionável de R_j sofreu alterações e D está no estado permanente. O usuário deve ser consultado para que decida realmente (ou uma opção padrão do sistema deve indicar) pela criação de um novo objeto de dados D_j correspondente a R_j em C_i, ou pela desistência das alterações em R_j, que passa para o estado permanente sem alterações. Caso decida pela criação, D_j é colocado no estado permanente e incluído na OD-base privada e no contexto de versões correspondente como derivado de D. O nó D_j deve ter, como conjunto de descritores alternativos, o mesmo conjunto de descritores alternativos de D. R_j é removido do contexto de variantes como derivado de D e inserido no novo contexto de variantes como derivado de D_j. Então tem-se dois casos a tratar:

i) D_j será incluído em todas as composições C_is relacionados a C'_is tais que R_j está contido em C'_i, se C_i estiver no estado temporário. Uma nova consulta deve ser

feita ao usuário para que ele decida pela remoção ou não de *D* em C_i, e em caso de não remoção, se quer transferir todas as referências a *D*, contido nessas composições C_is e que têm o mesmo descritor usado para criar R_j, para D_j.

i.1) Caso se decida pela remoção, qualquer referência por elo ou composição a *D*, contido nessas composições C_is, deve ser alterada para se referir a D_j em substituição. Todos os contextos de variantes devem ser atualizados de forma que todos os nós R_j, derivados de *D* e contidos em C'_i, passem agora a derivar de D_j.

i.2) Caso se decida pela não remoção e não se queira transferir todas as referências a *D* (que têm o mesmo descritor usado para criar R_j) para D_j, nessas composições C_is, se R_j é o nó de representação criado na navegação em profundidade quando se seleciona *D*, o conjunto de descritores alternativos associados ao nó *D* em C_i para criação de R_j deve ser duplicado e agora fazer parte da lista de conjunto de descritores de D_j. *D* deve ser incluído nos contextos C'_is correspondentes aos C_is (se lá não já estiver incluído) e todos os elos visíveis por R_j devem ter este nó substituído por *D*.

i.3) Caso se decida pela não remoção e se queira transferir todas as referências a *D* (contido nessas composições C_is e que têm o mesmo descritor usado para criar R_j) para D_j, nessas composições C_is, se R_j é o nó de representação criado na navegação em profundidade quando se seleciona *D*, o conjunto de descritores alternativos associados ao nó *D* em C_i para criação de R_j deve ser removido e agora fazer parte da lista de conjunto de descritores de D_j. Nas composições C_is, os elos visíveis por *D* em P_i, que tinham *D* como nó âncora (de destino ou fonte) e o descritor usado para criar R_j, devem agora ser apropriadamente direcionados para D_j. Todos os elos visíveis por *D* em P_i que têm *D* como âncora e um descritor diferente do descritor usado para criar R_j devem permanecer inalterados. Todos os demais elos visíveis por *D* em P_i que têm *D* na lista de nós de um ponto terminal devem ter *D* substituído por D_j em seus pontos terminais.

ii) Em todas as composições C_is relacionadas a C'_is tais que R_j está contido em C'_i e em que C_i estiver no estado permanente, tem-se um caso típico de propagação de versão na OD-base privada. Neste caso, deve-se percorrer a perspectiva P_i, de cada C_i em direção à OD-base privada, até que se encontre o primeiro contexto no estado temporário ou até que se chegue na base privada. Seja N_k o contexto encontrado na perspectiva P_i. É então criado em N_k um contexto no estado temporário N^*_{k-1} idêntico ao contexto N_{k-1} da perspectiva. No contexto de versões apropriado, N^*_{k-1} é inserido como derivando de N_{k-1}. De forma análoga, o nó N_{k-2} contido em N^*_{k-1} é substituído por um nó no estado temporário N^*_{k-2} idêntico a N_{k-2}, e assim sucessivamente até o nó C_i, inclusive, que será substituído pelo nó C^*_i. Nos contextos de versões apropriados, N^*_{k-i} (I>2) é inserido como derivando de N_{k-i}. Todas as referências por composição ou elo recursivamente contidas em N_k devem ser modificadas para refletir os novos nós criados. Uma nova perspectiva P^*_i é construída a partir de P_i, substituindo os nós N_{k-i} pelos nós N^*_{k-i} correspondentes. Todos os nós N_{k-i}s inseridos nos contextos de P'_i da OR-base privada devem ser substituídos pelos N^*_{k-i}s correspondentes. Todos os nós de representação derivados dos N_{k-i}s passam agora a ser derivados dos N^*_{k-i}s correspondentes e devem ter os respectivos contextos de variantes alterados para refletir a nova condição. O procedimento passa agora a ser idêntico ao caso i), onde C_i é substituído por C^*_i.

3) R_j não sofreu alterações e *D* está no estado temporário. Neste caso *D* simplesmente é tornado permanente.

4) R_j não sofreu alterações e *D* está no estado permanente. Neste caso nada mais é realizado.

2.3.4.3 Check-in

Um usuário pode mover um nó terminal ou de contexto de usuário de uma OR-base privada para uma OD-base privada ou de uma OD-base privada para a hiperbase

pública através da primitiva de *check-in*, desde que o nó esteja no estado permanente. Se um nó permanente de contexto de usuário *C* tem de ser movido, então, primeiramente, todos os nós contidos em *C* devem ser movidos para a mesma base.

No caso de *Check-in(D)* de um **objeto de dados** *D*, realmente ocorre uma mudança de base, isto é, o nó objeto de dados migra para a hiperbase pública tornando-se um nó objeto de armazenamento. Note que a movimentação de uma versão para a hiperbase pública só precisa ser realizada quando ocorrer alguma modificação no conteúdo original. Se após a criação de uma versão de dados *D* de um nó *A* da hiperbase pública em uma OD-base privada, *D* não sofrer qualquer modificação, *D* sofre a operação de *delete*, a ser definida na Seção 2.3.4.6, pois não há a necessidade de replicação de nós. Os nós de composição que contêm *D*, tanto na OD-base privada quanto na OR-base privada, bem como os elos, devem ser atualizados para conter agora *A*, ao invés de *D*. Da mesma forma, todas as versões criadas a partir de *D* devem ser transformadas em versões de *A* no contexto de versões apropriado.

Quando um objeto de dados sofre *check-in*, todos os objetos de representação dele derivados devem sofrer *check-in*.

Então, no caso do *Check-in(R)* de um nó **objeto de representação** *R*, ele é simplesmente tornado permanente e depois (tornado obsoleto e) destruído[11]. O objeto de dados correspondente *D* é inserido nas composições onde estava o nó de representação, se lá não estiver ainda inserido. Os elos da OR-base privada devem ser apropriadamente modificados para conterem *D* em substituição a *R*.

A execução do *check-in*, tanto em objetos de dados, como em objetos de representação, dispara notificações, como será visto no Capítulo 3.

---

[11] Observe que a mudança de base de fato não ocorre diretamente no *check-in*. Na verdade, a operação de mudança do estado de *R* para permanente efetuará (possivelmente) a criação de uma versão de dados correspondente à *R*, como visto na Seção 2.3.4.2, e a mudança de permanente para obsoleto, sua eventual destruição.

2.3.4.4 Move

Todos os nós contextos de usuário e nós terminais de uma OD-base privada *ODBP* podem ser movidos em bloco para a hiperbase pública através da primitiva *move*. Neste caso, diz-se que a OD-base privada foi movida para a hiperbase pública. Quando a primitiva é aplicada, todos os nós terminais e de contexto de usuário da base privada vão para o estado permanente e são transferidos para a hiperbase pública (através da primitiva *check-in*); e todas as bases privadas contidas em *ODBP* são recursivamente movidas para a hiperbase pública. No final do processo, a OD-base privada conterá apenas anotações (e elos associados) e bases privadas, que conterão apenas anotações e bases privadas, recursivamente.

2.3.4.5 Shift

Analogamente, todos os nós de contexto de usuário e nós terminais de uma OR-base privada *ORBP* podem ser movidos em bloco para uma OD-base privada através da primitiva *shift*. Neste caso diz-se que a OR-base privada foi movida para a OD-base privada. Quando a primitiva é aplicada, todos os nós terminais e de contexto de usuário da base privada vão para o estado permanente e são transferidos para a OD-base privada (através da primitiva *check-in*); e todas as bases privadas contidas em *ORBP* sofrem, recursivamente, a operação *shift*. No final do processo, a OR-base privada *ORBP* conterá apenas anotações (e elos associados) e bases privadas, que conterão apenas anotações e bases privadas, recursivamente.

2.3.4.6 Delete

Um usuário pode remover um nó *N* de uma base privada *BP* através da primitiva *delete*.

Se *N* é uma versão de dados no estado temporário, ele é removido do contexto de versões apropriado e então destruído. Todos os objetos de representação derivados de *N* devem também sofrer a aplicação da primitiva *delete* nas OR-bases privadas correspondentes. Análogo ao caso da remoção de uma versão de representação temporária, uma notificação será enviada aos nós que referenciam, por inclusão e por elo, o nó removido, como será visto no Capítulo 3. Em todas essas bases, o nó *N* deve

então ser removido de todas as composições que o contém, e todos os pontos terminais que têm *N* em sua lista de nós devem também ser removidos, devendo os pontos de encontro dos elos correspondentes também serem apropriadamente atualizados. Caso após a remoção de um ponto terminal o elo fique inconsistente (sem destino ou sem origem), o elo deve ser destruído.

Se o nó *N* é uma versão de dados no estado permanente, ele é tornado obsoleto. Após um nó se tornar obsoleto em uma OD-base privada, ele é movido para a hiperbase pública. Se houver a criação de um novo nó na hiperbase pública correspondente ao nó obsoleto movido, o nó na hiperbase pública deve estar no estado obsoleto. Se não houver criação de novo nó, o nó objeto de armazenamento correspondente deve trocar seu estado para obsoleto, se e somente se o usuário tiver direito de escrita no seu estado. Se *N* é um nó de contexto de usuário obsoleto, todos os seus nós componentes são transferidos também. Como no caso anterior, todos os objetos de representação derivados de *N* devem também sofrer a aplicação da primitiva *delete* nas OR-bases privadas correspondentes. Uma notificação será enviada aos nós que referenciam *N* por inclusão e por elo, como será visto no Capítulo 3. Em todas essas bases, *N* representa agora um objeto de armazenamento.

Se *N* é uma versão de representação no estado temporário, ele é removido do contexto de variantes apropriado e então destruído. Uma notificação será enviada aos nós que referenciam *N* por elo ou inclusão, como será visto no Capítulo 3. O nó *N* deve então ser removido de todas as composições que o contém, e todos os pontos terminais que têm *N* em sua lista de nós devem também ser removidos, devendo os pontos de encontro dos elos correspondentes também serem apropriadamente atualizados. Caso após a remoção de um ponto terminal o elo fique inconsistente (sem destino ou sem origem), o elo deve ser destruído.

Se *N* é uma versão de representação no estado permanente, ele é removido do contexto de variantes apropriado e então destruído. O nó *N* deve então ser substituído em todas as composições que o contêm, e em todos os pontos terminais que o têm em sua lista de nós, pelo nó objeto de dados de onde *N* derivou.

Se *N* é um objeto de armazenamento, ele deve passar para o estado obsoleto. Quando a primitiva *delete* é aplicada a partir de uma base privada objeto de dados *ODBP*, uma notificação será enviada aos nós que referenciam *N* por inclusão e por elo, como será visto no Capítulo 3. Em todas essas bases, o nó *N* deve então ser removido de todas as composições que o contém, e todos os pontos terminais que têm *N* em sua lista de nós devem também ser removidos, devendo os pontos de encontro dos elos correspondentes também serem apropriadamente atualizados. Caso após a remoção de um ponto terminal o elo fique inconsistente (sem destino ou sem origem), o elo deve ser destruído. Caso a primitiva *delete* seja aplicada a partir de uma base privada objeto de representação *ORBP*, ela é passada a *ODBP* de onde a *ORBP* foi derivada, passando-se tudo como no caso anterior.

Uma base privada *BP* também pode ser removida. Neste caso, todos os seus nós, incluindo bases privadas e registros correspondentes nos contextos de versões, são também removidos, primeira e recursivamente. A base privada *BP* é então destruída.

3. Controle de Notificação

3.1 Introdução

O controle de notificação permite que usuários do sistema sejam notificados de eventos importantes ocorridos em objetos de um documento, tais como: a criação de novas versões de um determinado nó, atualização de atributos, obsolescência de nós, remoção e adição de elos. Com o controle de notificação, o estado de edição de um documento pode ser divulgado para todos os autores.

O objetivo deste capítulo é apresentar um modelo de notificação para o NCM que irá agir em conjunto com o controle de versões descrito anteriormente, fornecendo um suporte à autoria cooperativa de documentos hipermídia.

Na Seção 3.2, alguns conceitos e princípios pertinentes ao controle de notificação serão discutidos. Um modelo de notificação para o NCM que tem como objetivos principais a extensibilidade, adaptabilidade e expressividade será apresentado na Seção 3.3.

3.2 Conceitos e Princípios

Uma notificação é uma unidade lógica de informação enviada entre duas entidades que tem como intuito informar a ocorrência de um determinado evento [REINERT 1998]. As notificações relevantes para esta dissertação são aquelas referentes a eventos relacionados à autoria cooperativa de hiperdocumentos, portanto, o termo notificação será utilizado para se referir a tais notificações.

Uma notificação pode ser classificada como imediata ou deferida, dependendo se as notificações são enviadas aos usuários imediatamente após a ocorrência do evento que disparou a notificação (evento de disparo) ou em algum momento depois.

De acordo com [BATISTA 1994] e [IBEX 1995], existem duas técnicas relativas à estratégia de notificação:

1. Baseada em Mensagens (*Message-Based*) – É uma técnica de notificação ativa, onde uma mensagem que relata a ocorrência de um determinado evento em alguma entidade *N* do sistema (usualmente nós) é enviada aos interessados nesse acontecimento. Para o usuário final, uma mensagem pode ser percebida como um *email*, um aviso sonoro, uma apresentação de uma janela no sistema contendo a notificação, etc.

2. Baseada em Sinais (*Flag-Based*) – É uma técnica de notificação passiva. Quando ocorre um determinado evento em alguma entidade *N* do sistema (usualmente nós), uma mensagem relatando a ocorrência será apresentada quando usuários acessarem essa entidade.

A estratégia baseada em mensagens provê uma interação maior com os autores, proporcionando uma visão mas precisa do estado de edição de um documento, sendo a mais adequada para ambientes de edição cooperativa de hiperdocumentos.

A referência [REINERT 1998] define os conceitos nível e granularidade de notificação. O nível de notificação especifica quais alterações em objetos realizadas por autores devem ser notificadas, enquanto que a granularidade define quais alterações podem disparar uma notificação. Observe a diferença entre os dois conceitos: o nível indica quais as alterações, dispostas pela granularidade, devem ser notificadas. Capturar modificações em atributos de objetos é considerado uma granularidade fina.

Para registrar as notificações enviadas a uma entidade, existe o *log*. O *log* de notificações e o contexto de versões de um nó proporcionam informações importantes sobre uma entidade ao longo do tempo, podendo auxiliar, inclusive, na operação de fusão de nós. O *log* também permite que o usuário consulte as notificações recebidas, caso as tenha ignorado a princípio.

3.3 Modelo de Notificação para o NCM

A especificação do mecanismo de notificação para o NCM foi baseada no paradigma *Publish/Subscribe* [MATHUR 1995]. A idéia essencial desse paradigma é relativamente simples: um ou mais editores (*publishers*) enviam dados para múltiplos

assinantes (*subscribers*). Tipicamente, os assinantes se inscrevem em determinados canais para receber mensagens publicadas pelos editores nesse canais.

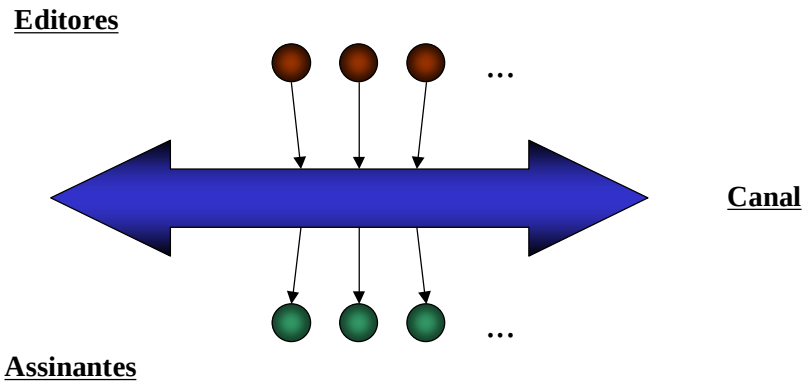


Figura 11: *Publish/Subscribe*

Dessa forma, no NCM, um nó N_a se inscreve para receber notificações da ocorrência de um determinado evento de um outro nó N_b. Por exemplo, uma versão *V'*, derivada de um nó *V*, pode se inscrever para receber notificações sempre que *V* sofrer uma atualização em qualquer de seus atributos não versionáveis.

O nó é a entidade do NCM que participa mais ativamente no mecanismo de notificação, publicando e recebendo notificações. Dois pontos relevantes, que são os mais importantes para a concepção de um esquema de notificação, motivam tal comportamento no controle de notificação: o versionamento e a edição compartilhada de objetos entre usuários. Ambos ocorrem em nós (ou em subclasses de nós) no NCM.

Os elos não publicam ou recebem notificações, apesar de atuarem nas regras de inscrição, como será visto na Seção 3.3.5. No caso de eventos relacionados a um elo *E*, optou-se por notificar o nó de contexto *C* que contém *E*, já que *C* é a entidade que determina realmente a participação de *E*.

Por enquanto, não foi julgado necessário o acréscimo de outras entidades do NCM como descritor ou âncora (além do elo) para publicar ou receber notificações. O motivo é que não há edição compartilhada ou versionamento em tais entidades, na versão atual do modelo.

Todavia, a adição de novas entidades ao mecanismo de notificação seria relativamente simples. Para isso, as novas entidades poderiam pertencer à nós de contexto de notificação [Seção 3.3.2] e à nós *containers* de notificação [Seção 3.3.3]; os atributos inseridos na classe nó [Seção 3.3.4] teriam de ser replicados para as novas entidades; e, caso necessário, novas regras de inscrição e outros eventos de disparo teriam de ser criados.

Como comentado anteriormente, os nós são as entidades que publicam ou recebem notificações. Contudo, o sistema que utilizará o mecanismo de notificação do NCM também pode capturar as notificações enviadas a um nó e realizar as ações desejadas.

O controle de notificação fornece um mecanismo automático de adição e remoção de inscrições, de acordo com regras, inicialmente definidas [Seção 3.3.5], que refletem as necessidades básicas de notificação de um sistema de autoria hipermídia com controle de versões. O modelo permite ativar ou desativar cada uma dessas regras de inscrição e possibilita que o usuário adicione e remova suas próprias inscrições [Seção 3.3.2].

Assim, no modelo, regras geram inscrições automáticas para nós, e essas inscrições permitem a recepção das notificações propriamente ditas, disparadas na ocorrência de eventos.

O esquema de notificação foi especificado para permitir uma maior extensibilidade, adaptabilidade e expressividade no modelo de notificação. Extensibilidade pelo fato de que as regras que irão determinar as inscrições podem ser vistas como componentes que podem ser incluídos ou excluídos à medida do necessário. Adaptabilidade porque, como será visto adiante, as regras automáticas de inscrição oferecidas pelo modelo podem ser ativadas ou desativadas, moldando-se de acordo com os requisitos de um sistema de autoria hipermídia que utilizará esse modelo. Expressividade porque as notificações entre nós podem ser especificadas a partir de inscrições que capturam as interações desejadas entre os nós.

É bom observar que o tratamento de notificações entre nós independe dos planos (armazenamento, dados ou representação) a que esses nós pertencem, ou seja,

o tratamento é único para uma versão de representação derivada a partir de um objeto de dados ou para uma versão de dados derivada a partir de um objeto de armazenamento ou de um objeto de dados. Portanto, o termo versão, neste capítulo, se refere de maneira indistinta a uma versão de representação ou a uma versão de dados.

3.3.1 Eventos de Disparo

Por capturar modificações em atributos, a granularidade de notificação no NCM é classificada como fina. Ela é dada pelos eventos de disparo que ocorrem em nós do NCM, e que podem gerar notificações[12] consideradas relevantes ao suporte à autoria cooperativa em sistemas hipermídia. Vários desses eventos já foram identificados no Capítulo 2, e são aqui resumidos.

1. Derivação de Versão

Esse evento de disparo representa tanto a derivação de versão por meio da primitiva *check-out*, como pela inclusão manual (usuário) de um elo em um contexto de versões. Será útil para informar que versões de um nó estão sendo criadas, caracterizando uma provável[13] edição paralela. Esse é um dos eventos de grande utilidade para o mecanismo automático de inscrições, descrito na Seção 3.3.5.

2. *Check-in*

É o movimento de um nó de uma OD-base privada para a hiperbase pública ou de uma OR-base privada para uma OD-base privada. Fornecerá a informação de que novas versões de um nó estão disponíveis na hiperbase pública ou em uma OD-base privada.

---

[12] Além de dispararem notificações, alguns desses eventos de disparo irão gerar inscrições automáticas, conforme será visto na Seção 3.3.5.

[13] Provável porque a nova versão não será necessariamente editada, pode acontecer que seja somente lida.

3. Estabilidade

É a mudança do estado de temporário para permanente. Útil para informar que um determinado nó alcançou um estado consistente, estável.

4. Obsolescência

É a mudança do estado de permanente para obsoleto. Útil para informar que um determinado nó chegou em um estado obsoleto e está em desuso.

5. Inserção de um Nó

É a operação de inserção de um nó em uma composição. Por exemplo, pode ser usado para capturar alterações em uma base privada *BP*, informando que existe um novo nó disponível em *BP*.

6. Remoção de um Nó

É a exclusão de um nó em uma composição. Útil para notificar as referências por inclusão e por elo nas operações de *delete* em nós de uma base privada, como mencionado na Seção 2.3.4.6.

7. Atualização de Atributos Não Versionáveis

É a atualização de um dos atributos não versionáveis de um nó, esteja ele na hiperbase pública ou em alguma base privada. É interessante observar que um nó pode receber, além da notificação do evento de atualização ocorrido, o próprio conteúdo da atualização efetuada. Para isso, a lista de atributos genéricos da mensagem de notificação [Seção 3.3.4] pode ser utilizada para o transporte da atualização.

8. Atualização de Atributos Versionáveis

É a atualização de um dos atributos versionáveis de um nó temporário. Nós permanentes só podem ter seus atributos versionáveis alterados em uma nova versão temporária.

9. Inserção de Elo

É a inserção de um elo em um nó de contexto de usuário, contexto de versões, base privada ou base de contexto de versões. Esse é outro evento de disparo que atuará junto ao mecanismo automático de inscrição, descrito na Seção 3.3.5.

10. Remoção de Elo

É a exclusão de um elo em um nó de contexto de usuário, contexto de versões, base privada ou base de contexto de versões. Esse é um evento de disparo que atuará junto ao mecanismo automático de cancelamento, descrito na Seção 3.3.6.

11. Adição de Novos Atributos

No NCM, novos atributos podem ser adicionados a um nó. Esse é o evento de disparo que captura essa operação.

12. Recebimento de Notificação

Este evento de disparo ocorre quando um nó *N* recebe uma notificação, sendo identificado por um par (*O, E'*), onde:

O é a relação de parentesco de *N* com o remetente da notificação, o nó *P*, podendo assumir os valores:

Ancestral – P é um nó ancestral de *N* no grafo de derivação.

Ascendente – P é um nó ascendente de *N* no grafo de derivação.

Descendente – P é um nó descendente de *N* no grafo de derivação.

Independente – P é um nó sem qualquer relação de derivação com *N*.

E' é um evento de disparo, incluindo o próprio evento *recebimento de notificação*.

Dessa maneira, um evento de disparo deste tipo é representado como *Recebimento de Notificação* (*O*, *E'*).

Por exemplo, suponha que o nó *V* deriva *V'*. Caso *V* seja notificado da estabilidade de *V'*, o evento de disparo *Recebimento de Notificação* (*Descendente*, *Estabilidade*) é dado como ocorrido em *V*.

O motivo de captura desse evento é explicado logo abaixo.

13. Destruição

Ocorre quando um nó *N* é destruído, ao ser deletado pelo usuário quando está no estado temporário ou removido pela coleta de lixo quando está no estado obsoleto. Será útil também para comunicar o cancelamento de inscrições referentes aos contextos de notificação de *N*, como será visto na Seção 3.3.6.5.

Dados esses eventos, a Tabela 2 mostra quais deles podem efetivamente ocorrer em uma determinada classe derivada de nó.

É bom ressaltar que o *Check-out* (no caso 1) e o *Check-in* somente serão considerados quando houver efetivamente a derivação de uma nova versão ou o movimento de um nó entre bases, respectivamente. Como foi visto no Capítulo 2, a primitiva *Check-out (P, C_{ont})* não deriva versão quando já existe uma versão do nó recursivamente contido no contexto *Cont* e o *Check-in* não efetua a movimentação do nó quando não existem modificações em relação à versão original.

	Terminal	Contexto de Usuário	Hiperbase Pública	Base Privada	Base Cont. Versões	Cont. de Versões	Cont. de Variantes	Trilha
Derivação Versão	✓	✓						
Check-in	✓	✓						
Permanência	✓	✓						
Obsolescência	✓	✓						
Inserção Nó		✓	✓	✓	✓	✓	✓	✓
Remoção Nó		✓	✓	✓	✓	✓	✓	✓
Atual. Atrib. Não Versionável	✓	✓	✓	✓	✓	✓	✓	✓
Atual. Atrib. Versionável	✓	✓						
Inserção de Elo		✓		✓	✓	✓		
Remoção de Elo		✓		✓	✓	✓		
Adição Novos Atributos	✓	✓	✓	✓	✓	✓	✓	✓
Recebimento de Notificação	✓	✓	✓	✓	✓	✓	✓	✓
Destruição	✓	✓	✓	✓	✓	✓	✓	✓

Tabela 2: Possibilidade de ocorrência de eventos em nós NCM

O objetivo do evento *recebimento de notificação* é permitir o repasse de notificações entre nós. Por exemplo, suponha que *V* deriva *V'* que por sua vez deriva *V''*, como mostrado na Figura 12.

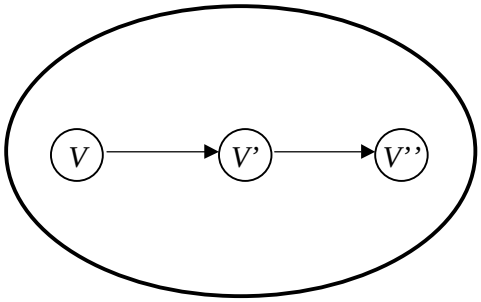


Figura 12: Contexto de versões de *V*

Caso ocorra a obsolescência de *V*, pode-se desejar que, automaticamente, os filhos de derivação e os filhos dos filhos (netos) de derivação de *V*, ou seja, *V'* e *V''* sejam notificados. Dessa maneira, *V* notifica diretamente *V'*, este recebe a notificação e a repassa para *V''*, utilizando esse evento de disparo. Assim, é possível que ocorra a situação em que *V* é tornado obsoleto e, ainda assim, *V''* ser notificado, mesmo que *V'* não sofra alteração.

O evento de repasse também será útil para a notificação entre vizinhos no grafo de derivação, como será visto na Seção 3.3.5.

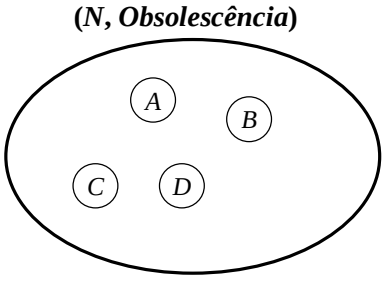
3.3.2 Contexto de Notificação

Para identificar que nós devem receber notificações da ocorrência de um evento de um determinado nó, uma nova entidade foi adicionada ao modelo NCM: o nó de contexto de notificação.

Um *nó de contexto de notificação CN* é um nó de contexto cujo conteúdo é um conjunto *S* de nós[14], exceto nós de contexto de notificação. O nó *CN* possui dois atributos adicionais: o *UID* de um nó *N*, que não pode ser um nó de contexto de notificação, e um evento de disparo *E*, que pode ser um dos eventos descritos na Seção 3.3.1. Diz-se que um nó *N* em *S* é um *componente de CN* e que *N* está contido em *CN*. Diz-se também que *N* está inscrito em (ou assina) *CN*, e *CN* é o contexto de notificação (*N*, *E*).

CN assume o papel de um canal no paradigma *Publish/Subscribe*, onde o nó *N* (editor) publica notificações em *CN*, que são recebidas por outros nós (assinantes), componentes de *CN*. Ou seja, na ocorrência do evento *E* em *N*, os componentes de *CN* são notificados.

Por exemplo, a Figura 13 apresenta o contexto de notificação (*N*, *Obsolescência*), onde os nós *A*, *B*, *C* e *D* irão receber uma notificação quando o nó *N* tornar-se obsoleto.



[14] Será visto na Seção 3.3.3 que *S* poderá conter nós de uma nova classe denominada *container* de notificação.

Figura 13: Nó de contexto de notificação contendo os nós *A, B, C* e *D*

A Figura 14 apresenta o contexto de notificação (*N*, *Recebimento de Notificação*(*Ascendente*, *Obsolescência*)), onde os nós *X*, *Y* e *Z* irão receber uma notificação quando o nó *N* receber uma notificação de que um de seus ascendentes no grafo de derivação tornou-se estável.

(*N, Recebimento de Notificação(Ascendente, Estabilidade*))

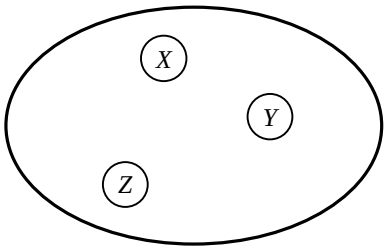


Figura 14: Nó de contexto de notificação contendo os nós *X, Y* e *Z*

Dado um nó *N*, o Apêndice B apresenta todos os contextos de notificação referentes a *N*, que estão disponíveis para inscrições (manuais) realizadas por usuários.

Para facilitar esse processo de inscrições avulsas realizadas por usuários, um determinado nó deve ser capaz de se inscrever em um outro nó qualquer, sem depender de parâmetros além do nó e o evento de disparo a serem observados. É por esse motivo que um contexto de notificação deve ser simplesmente identificado pelos dois referidos parâmetros.

Para ser inscrito em um determinado contexto de notificação *CN*, um nó *A* deve ser inserido na lista de nós de *CN*, e para cancelar a inscrição, *A* deve ser removido da lista de nós de *CN*.

Suponha a inscrição de um nó *A* no contexto de notificação (*N*, *E*), onde *E* é um evento diferente de destruição. Caso *A* não pertença ao contexto de notificação (*N*, *Destruição*), *A* deve ser inserido automaticamente nesse contexto.

Suponha o cancelamento da inscrição de um nó *A* no contexto de notificação (*N*, *E*), onde *E* é um evento diferente de destruição. Caso *A* não pertença a outro contexto de notificação de *N*, *A* deve ser removido automaticamente do contexto de notificação (*N*, *Destruição*).

A inclusão e exclusão automáticas de nós do contexto de notificação de um nó relativo ao evento de destruição será útil para o mecanismo automático de cancelamento que será visto na Seção 3.3.6.

A criação e destruição de nós de contexto de notificação devem ser realizadas sob demanda. Ou seja, dado um nó *N* e um evento de disparo *E*, na primeira solicitação de inscrição no evento *E* de *N*, deve-se criar o contexto de notificação (*N*, *E*). No caso de remoção (cancelamento) do último nó de um contexto de notificação *CN*, *CN* deve ser destruído.

No caso de destruição de um nó *N*, *N* publica uma notificação no contexto de notificação (*N, Destruição*) [Seção 3.3.6.5] e todos os contextos de notificação referentes à *N* devem ser destruídos.

3.3.3 *Container* de Notificação

Em um mecanismo de notificação, um fator importante a ser considerado é a privacidade. O modelo de notificação deve permitir que, dado um agrupamento *G* de nós formado eventualmente, notificações possam ser trocadas por membros de *G* de maneira privativa.

Por exemplo, um agrupamento de nós pode ser formado por um subconjunto de versões irmãs no grafo de derivação de um determinado nó, que representam a autoria cooperativa privativa de um determinado documento.

Para facilitar a implementação desse conceito de privacidade para notificação em agrupamentos, uma nova entidade foi adicionada ao modelo NCM: o *container* de notificação.

Um *container de notificação G* é um nó de contexto cujo conteúdo é um conjunto *S* de nós, exceto nós de contexto de notificação. Diz-se que um nó *N* em *S* é um *componente de G* e que *N está contido em G*. *G* pode conter outros *containers* de notificação, e pode pertencer a mais de um *container* de notificação, no entanto, *G* somente pode pertencer a nós de contexto de notificação e a *containers* de notificação.

G possui adicionalmente um conjunto *F* de filtros, que determinam o comportamento da recepção de notificações dos componentes de *G* e de inserções de nós em *G*.

Assim, um *container* de notificação *G* pode se inscrever em um contexto de notificação *CN*, fazendo com que os nós pertencentes a *G* recebam, de acordo com filtros em *F*, as notificações publicadas em *CN*.

Suponha o exemplo da Figura 15, em que o nó de contexto de notificação (*N*, *Derivação de Versão)* possui os nós *A*, *B*, *C*, *D* e o *container* de notificação *G*, que por sua vez possui os nós *X*, *Y* e *Z*. Suponha também que existe um único filtro em *G* que indica que derivações de versões em uma base privada *BP* só devem ser comunicadas ao nó *Z*.

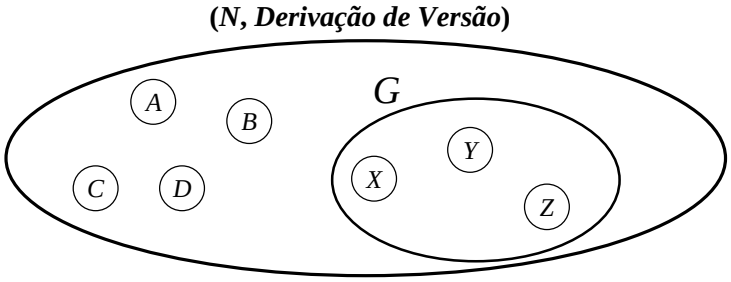


Figura 15: Contexto de notificação contendo o *container* de notificação *G*

Nesse exemplo, quando *N* derivar uma versão, os nós *A*, *B*, *C*, *D* e *G* serão notificados. Caso a versão derivada pertença à *BP*, somente *Z* será notificado, e caso não pertença, *X*, *Y* e *Z* serão notificados.

Para exemplificar a possibilidade de troca de notificações privativas, considere o exemplo da Figura 16. Suponha que *N* deriva as versões *A*, *B*, *C* e *D*, e que, além das inscrições apresentadas nessa figura, *N* está inscrito no evento de atualização de atributos versionáveis dos nós *A*, *B*, *C* e *D*. Suponha também que existe um único filtro em G_1 indicando que somente devem ser repassadas notificações provenientes dos nós *A* e *B*, e outro único filtro em G_2 indicando que somente devem ser repassadas notificações provenientes dos nós *C* e *D*. Assim, quando o nó *A*, por exemplo, sofre uma atualização em qualquer de seus atributos versionáveis, *N* recebe uma notificação e repassa a notificação aos *containers* G_1 e G_2. G_1 repassa a notificação aos seus componentes, já que seu filtro permite (o remetente original da notificação é o nó *A*),

contudo, G_2 não repassa a notificação aos seus componentes, já que seu filtro bloqueia repasses de notificação onde o nó origem é diferente dos nós *C* e *D*.

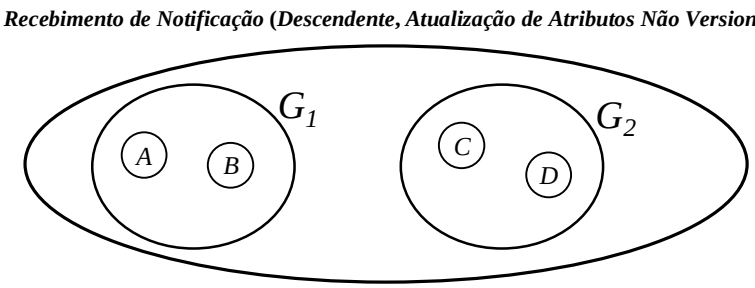


Figura 16: Contexto de notificação contendo os *containers* de notificação G_1 e G_2

A inserção de um nó em um *container* também é determinada por filtros, que podem, por exemplo, restringir a inclusão de nós pertencentes a uma determinada base privada.

É bom observar que, diferentemente dos nós de contexto de notificação, um *container* de notificação pode existir mesmo que não contenha qualquer componente.

3.3.4 A Classe Nó

É interessante que todas as classes derivadas (direta ou indiretamente) da classe *nó* participem do mecanismo, trocando notificações, porque os eventos relevantes para divulgação ocorrem em objetos dessas classes. Por exemplo, atualização de atributos de um nó terminal, inclusão de um nó na hiperbase pública, remoção de um elo em um nó de contexto de usuário, derivação de uma nova versão de um nó imagem, etc.

Como foi visto anteriormente, para receber notificações da ocorrência de um evento em um nó *N*, um nó se inscreve no contexto de notificações correspondente à *N* e ao evento de disparo desejado.

Para registrar as notificações recebidas por um determinado nó, o atributo *log* foi adicionado à classe nó. Esse atributo é formado pelos campos:

⇒ Data de Envio - Registra a data que a mensagem foi enviada.

⇒ Data de Recebimento - Registra a data que a mensagem foi recebida

⇒ Contexto de Notificação - Registra o contexto de notificação (nó, evento de disparo) em que a mensagem foi recebida.

⇒ Origem – Registra o nó que inicialmente disparou a notificação. Note que, dado o contexto de notificação, tem-se a identificação do nó que enviou a notificação e do evento de disparo. Contudo, o nó que enviou pode ser um nó de repasse, logo, é interessante que o nó original que disparou a notificação seja registrado.

Esse registro das notificações coletadas ao longo do tempo pode ser consultado a qualquer momento, fornecendo informações úteis que podem ter passado despercebidas pelo usuário.

A primitiva *notify* (*CN*, *M*) permite que um determinado nó publique notificações em um contexto de notificação, onde *CN* é um contexto de notificação e *M* uma mensagem de notificação. Uma *mensagem de notificação* possui uma lista *L* de atributos genéricos e pode ser utilizada para transportar informações como data de envio e identificador do nó remetente. Diz-se que um atributo *A* pertencente à lista *L* de uma mensagem de notificação *M* é um componente de *M* e que *M* contém *A*.

Para cada evento de disparo *E* que ocorre em um nó *N*, automaticamente uma mensagem de notificação *m* contendo a data em que ocorreu o evento é criada, e a primitiva *notify* (*CN*, *M*) é invocada com *CN* igual ao contexto de notificação (*N*, *E*), e *M* igual a *m*, caso o contexto de notificação (*N*, *E*) exista[15]. Assim, por exemplo, o nó *N*, ao ser tornado obsoleto, publica no contexto de notificação (*N*, *obsolescência*) uma mensagem de notificação contendo somente a data de envio. Os nós pertencentes ao contexto de notificação (*N*, *obsolescência*) receberão a mensagem de notificação e perceberão, portanto, que *N* tornou-se obsoleto.

[15] Repare que não é necessária a invocação de tal primitiva quando não existir o contexto de notificação (*N*, *E*), já que, com certeza, não existem nós inscritos no evento *E* do nó *N* (contextos de notificação são criados sob demanda).

A publicação em um contexto de notificação relacionado ao evento *recebimento de notificação* é um caso especial. Quando um nó *N* recebe uma notificação de um nó *P*, uma mensagem de notificação *M* é criada contendo a data de envio e o identificador do remetente *P*. Caso a notificação recebida seja referente a um evento de disparo *E* diferente do evento de *recebimento de notificação*, *E* deve ser incluído na mensagem de notificação *M* para que esse evento original de disparo seja identificado, no caso de seguidos repasses. Assim, *N* publica *M* no contexto de notificação (*N*, *recebimento de notificação*(*O, E'*)), onde *O* é a relação de parentesco (ancestral, ascendente, descendente ou independente) de *N* com o remetente da notificação, o nó *P*, como descrito na Seção 3.3.1, e *E'* um evento de disparo. Entretanto, há uma exceção: quando *E'* for um evento de recebimento de notificação e o valor de *O* for diferente de "ancestral", *N* não publicará a notificação. Repare que, nesse caso, existe uma restrição no repasse de notificações a uma distância de até dois nós. Essa restrição não interfere em qualquer das regras automáticas de inscrição do modelo [Seção 3.3.5], incluindo notificações entre nós que sejam vizinhos no grafo de derivação, e poda o repasse de notificações provavelmente irrelevantes.

Todavia, pode-se desejar que essa poda não seja realizada. Para isso, as inscrições em eventos de repasse podem ser efetuadas no contexto de notificação de recebimento de notificação de ancestrais, as quais são sempre repassadas.

3.3.5 Mecanismo Automático de Inscrição

O objetivo desse mecanismo é fornecer inscrições automáticas baseadas em regras definidas ao longo desta seção.

É importante levar em consideração quais regras devem existir no modelo, de modo a não gerar notificações desnecessárias e nem deixar de gerar notificações importantes. A princípio, um sistema hipermídia que utilizará o modelo pode adotar dois procedimentos (não excludentes). O primeiro é desativar as regras de inscrição que sejam consideradas desnecessárias ao sistema de autoria hipermídia e o segundo é acrescentar inscrições que não foram realizadas previamente pelo mecanismo automático de inscrição, porém consideradas importantes para o sistema. Deve haver

um compromisso entre esses dois pontos para que os problemas abaixo citados sejam reduzidos:

- A maioria dos sistemas de autoria desativam as mesmas regras de inscrição, fornecidas automaticamente pelo modelo de notificação do NCM.
- A maioria dos sistemas de autoria necessitam das mesmas inscrições, não fornecidas automaticamente pelo modelo de notificação do NCM.

Para atingir um conjunto de regras automáticas que seja considerado ótimo, ou seja, com regras que minimizam os dois problemas citados acima, é necessária a utilização sucessiva do modelo por vários sistemas de autoria cooperativa hipermídia, de modo a capturar com mais precisão as necessidades reais dos usuários em relação aos vários tipos de notificações ativas nesses sistemas. Com isso, o conjunto de regras automáticas oferecido inicialmente pelo modelo pode sofrer uma série de refinamentos para o alcande de um ponto ideal. A sugestão inicial do conjunto foi baseada em observações e experiências empíricas, tendo-se o intuito de fornecer notificações básicas e comuns a qualquer sistema de autoria hipermídia com controle versões.

Cada regra de inscrição automática pode ser ativada ou desativada no modelo, mas, inicialmente, estão todas ativas.

Observe que as regras automáticas não definem inscrições para *containers* de notificação. A funcionalidade de notificações privativas (inscrições em *containers*, e estes em contextos de notificação, por exemplo) deve ser solicitada pelo sistema hipermídia que utilizará o modelo de notificação, conforme o desejado.

A seguir, serão descritos alguns eventos que disparam inscrições automáticas, de acordo com regras pré-definidas.

3.3.5.1 Derivação de Versão

O evento de derivação de versão tanto dispara notificações, como gera inscrições. É importante que o procedimento de disparo de notificações seja realizado antes da geração de novas inscrições, para que uma versão recém derivada não receba mensagens referentes ao seu próprio evento de derivação.

Quando um nó *V* deriva uma versão *V'*, as inscrições determinadas pelas regras abaixo são efetuadas.

Regra 1

V' se inscreve no contexto de notificação (*V, derivação de versão*). Com isso, *V'* pode perceber quando uma nova versão de seu ascendente for criada, indicando a existência de uma provável edição paralela.

Regra 2

V' se inscreve no contexto de notificação (*V, atualização de atributos não versionáveis*). *V'* poderá então perceber que existem valores de atributos mais recentes que os seus, atualizados em seu ascendente, podendo então solicitar o novo valor do atributo, ao ser notificado.

Regra 3

V' se inscreve no contexto de notificação (*V, obsolescência*). É importante que um nó seja informado quando o nó do qual derivou tornou-se obsoleto e está em desuso.

Regra 4

V' se inscreve no contexto de notificação (*V, recebimento de notificação*(*ascendente, atualização de atributos não versionáveis*)). *V'* poderá então perceber que existem valores de atributos mais recentes que os seus, atualizados no ascendente de *V*, podendo então solicitar o novo valor do atributo, ao ser notificado.

Regra 5

V' se inscreve no contexto de notificação (*V, recebimento de notificação*(*ascendente, obsolescência*)). É importante que *V'* seja informado quando o nó do qual *V* derivou tornou-se obsoleto e está em desuso, independente se *V* será tornado obsoleto.

Regra 6

V' se inscreve no contexto de notificação (*V, recebimento de notificação*(*descendente, atualização de atributos não versionáveis*)). *V'* poderá então perceber que existem valores de atributos mais recentes que os seus, atualizados em algum de seus vizinhos no grafo de derivação, membro de uma edição paralela, podendo então solicitar o novo valor do atributo, ao ser notificado.

Regra 7

V' se inscreve no contexto de notificação (*V, recebimento de notificação*(*descendente, check-in*)). *V'* poderá então perceber que um de seus vizinhos no grafo de derivação, membro de uma edição paralela, foi disponibilizado na hiperbase pública ou em uma OD-base privada.

Regra 8

V' se inscreve no contexto de notificação (*V, recebimento de notificação*(*descendente, destruição*)). *V'* poderá então perceber que um de seus vizinhos no grafo de derivação, membro de uma edição paralela, foi destruído e portanto não participa mais dessa edição.

Regra 9

V se inscreve no contexto de notificação (*V', atualização de atributos não versionáveis*). Juntamente com a regra 6, o repasse de notificações aos vizinhos no grafo de derivação será feito por meio do ascendente no grafo de derivação.

Regra 10

V se inscreve no contexto de notificação (*V', check-in*). Juntamente com a regra 7, o repasse de notificações aos vizinhos no grafo de derivação será feito por meio do ascendente no grafo de derivação.

Regra 11

V se inscreve no contexto de notificação (*V'*, *destruição*). Juntamente com a regra 8, o repasse de notificações aos vizinhos no grafo de derivação será feito por meio do ascendente no grafo de derivação.

Para ilustrar as inscrições automáticas realizadas para o evento de derivação de versão (caso da regra 1), suponha que um nó *N*, inicialmente, derivou um nó *A*. *A* é inscrito no contexto de notificação (*N*, *derivação de versão*), como mostra a Figura 17.

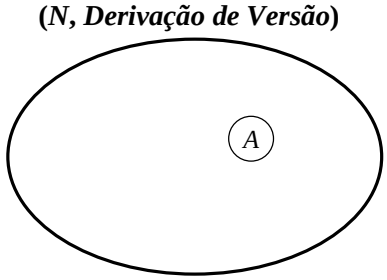


Figura 17: Contexto de notificação contendo o nó *A*

Suponha agora que *N* deriva uma nova versão *B*. Antes de novas inscrições serem geradas, uma notificação é publicada no contexto de notificação (*N*, d*erivação de versão*), fazendo com que o nó *A* seja notificado. Depois disso, *B* é inserido nesse mesmo contexto de notificação, como mostra a Figura 18.

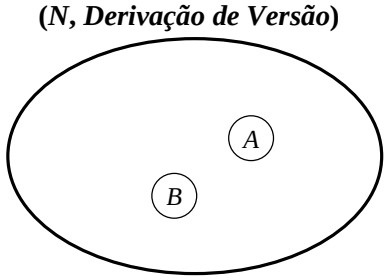


Figura 18: Contexto de notificação contendo os nós *A* e *B*

Caso *N* derive uma nova versão, o procedimento se repete: *A* e *B* são notificados e a nova versão é inserida no contexto de notificação (*N*, *derivação de versão*).

Em uma visão prática, a idéia dessa primeira regra é que um usuário que detenha uma versão de um determinado nó, por exemplo o texto da página principal na Internet do Laboratório Telemídia, seja notificado quando qualquer outro usuário criar

versões a partir deste nó, divulgando assim uma potencial edição cooperativa de um objeto, a princípio, de interesse comum.

De acordo com as regras acima, quando um nó *V* torna-se obsoleto ou sofre atualização em algum de seus atributos não versionáveis, é enviada uma notificação aos descendentes de *V* informando o evento. Cada descendente recebe a notificação e a repassa aos seus descendentes (regras 4 e 5). Esse é o comportamento padrão do modelo, nada impede que um sistema de autoria hipermídia desative as regras que geram esse repasse de notificações.

Um argumento a favor da inclusão dessas regras de repasse é o seguinte: suponha que um nó *V* deriva um nó *V'* que por sua vez deriva *V''*. Pela regra 2, a atualização de um atributo não versionável de *V* gera uma notificação para *V'* e pela regra 4, *V'* notificará por *V''*, mesmo que *V'* não tenha aprovado essas atualizações em atributos não versionáveis de *V*. Assim, *V''* pode aprovar as atualizações independentemente da aprovação de *V'*. Repare que se *V* sofre uma atualização de atributos não versionáveis, um nó *V'''*, derivado de *V''*, somente recebe a notificação caso *V'* ou *V''* aprovem a atualização. Esse é um efeito da estratégia de poda na distância do repasse, evitando que notificações provavelmente irrelevantes sejam propagadas.

Em alguns casos, existe a situação onde um nó *V* possui vários descendentes, onde esses descendentes precisam trocar notificações entre si. Tipicamente, um descendente informa todos os outros vizinhos no grafo de derivação a ocorrência de um determinado evento[16]. Para reduzir o número de inscrições, sempre que um descendente precisar notificar os outros vizinhos no grafo de derivação, ele o faz por meio do ascendente no grafo de derivação, utilizando o evento de disparo de recebimento de notificação. Por exemplo, suponha que *V* deriva V_1, V_2, V_3 e V_4 como mostrado na Figura 19.

[16] Útil, por exemplo, para indicar o início, término e estado de edição de um trabalho semi-síncrono em um sistema de autoria hipermídia com suporte à autoria cooperativa.

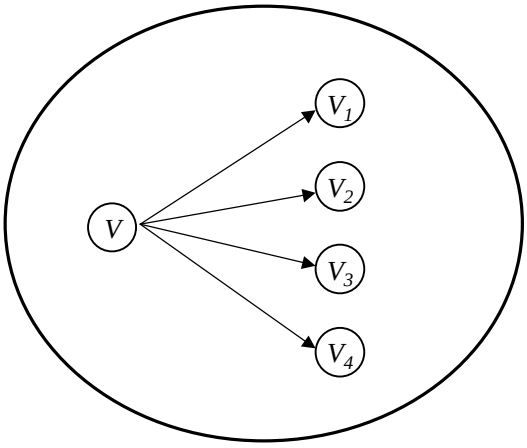


Figura 19: Contexto de versões de *V*

Caso os vizinhos no grafo de derivação tenham de trocar notificações entre si na ocorrência de um determinado evento, 12 inscrições (n*(n-1), onde n é o número de descendentes) tem de ser geradas ($V_1 \Rightarrow V_2$, $V_1 \Rightarrow V_3$, $V_1 \Rightarrow V_4$, $V_2 \Rightarrow V_1$, $V_2 \Rightarrow V_3$, $V_2 \Rightarrow V_4$, $V_3 \Rightarrow V_1$, $V_3 \Rightarrow V_2$, $V_3 \Rightarrow V_4$, $V_4 \Rightarrow V_1$, $V_4 \Rightarrow V_2$, $V_4 \Rightarrow V_3$), ao passo que com a notificação por meio do ascendente no grafo de derivação, 8 inscrições (n+n=2n, onde n é o número de descendentes) tem de ser geradas ($G \Rightarrow V_1$, $G \Rightarrow V_2$, $G \Rightarrow V_3$, $G \Rightarrow V_4$, $V_1 \Rightarrow G$, $V_2 \Rightarrow G$, $V_3 \Rightarrow G$, $V_4 \Rightarrow G$). Repare que um descendente pode receber a notificação de um evento que aconteceu com ele próprio, logo, é necessária uma restrição de que uma mensagem de notificação só é recebida, caso o nó remetente original e nó alvo sejam diferentes. Observe também que é possível que um usuário especifique diretamente uma notificação entre vizinhos no grafo de derivação.

3.3.5.2 Inserção de um nó *V* em um contexto de usuário

Quando um nó *V* é inserido em um nó de contexto de usuário *C*, a inscrição determinada pela regra abaixo é efetuada:

Regra 1

C se inscreve no contexto de notificação (*V*, *obsolescência*). Um nó de contexto de usuário será notificado quando um de seus nós internos tornar-se obsoleto.

3.3.5.3 Inserção de um elo em um nó de contexto de usuário

Seja *m* a quantidade de pontos terminais (origem ou destino) do elo. Seja $V_{i,}$ para $i \in [1,m]$, o nó âncora do ponto terminal *i*. Quando um elo é inserido em um nó de contexto de usuário *C*, as inscrições determinadas pelas regras abaixo são efetuadas:

Regra 1

C se inscreve nos contextos de notificação (V_i, *recebimento de notificação*(*descendente*, *check-in*)), onde $i \in [1,m]$. *C* pode perceber que existe uma versão mais recente de um nó que seu elo referencia.

Regra 2

C se inscreve nos contextos de notificação (V_i, *obsolescência*), onde $i \in [1,m]$. É importante que *C* possa perceber que um nó que seu elo referencia tornou-se obsoleto.

Regra 3

C se inscreve nos contextos de notificação (V_i, *remoção*), onde $i \in [1,m]$. *C* será alertado que um elo seu teve o nó âncora em um de seus pontos terminais removido.

No caso de adição de um ponto terminal a um elo *E* pertencente a um nó de contexto de usuário *C*, seja V_1 o nó âncora do ponto terminal. Cada uma das regras acima será executada com *i* sempre igual a 1.

3.3.6 Mecanismo Automático de Cancelamento

O objetivo desse mecanismo é preservar a consistência do modelo de notificação, removendo inscrições em eventos que não ocorrerão mais ou que se tornaram irrelevantes. As exclusões devem ser efetuadas apenas em inscrições realizadas pelo sistema: os contextos de notificação assinados pelos usuários devem ser preservados.

Deve haver um cuidado especial para remover as inscrições corretas, evitando que as necessárias ao sistema sejam removidas indevidamente. Por exemplo, suponha o caso da Figura 20, onde o nó de contexto de usuário *C* está inscrito no contexto de notificação (V_1, *obsolescência*) por dois motivos[17]: um por conter um elo que possui V_1 como nó âncora de um de seus pontos terminais [Seção 3.3.5.3, regra 2], e outro por conter V_1 diretamente [Seção 3.3.5.2, regra 1].

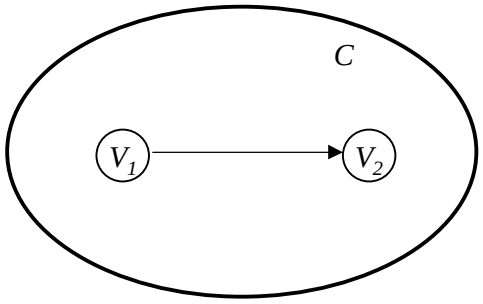


Figura 20: Nó de contexto de usuário *C* contendo dois nós

[17] Como não pode haver repetição de componentes em um nó de contexto, *C* está, a rigor, inscrito uma única vez no contexto de notificação (V_1, *obsolescência*).

Já no caso da Figura 21, o nó de contexto de usuário *C* está inscrito no contexto de notificação (*V₁*, *obsolescência*) por *V₁* ser um nó âncora de um ponto terminal do elo, e no contexto de notificação (*C₁*, *obsolescência*) por *C₁* estar diretamente contido em *C*. Nesse caso, são duas inscrições realizadas em contextos de notificação distintos, diferentemente do caso anterior.

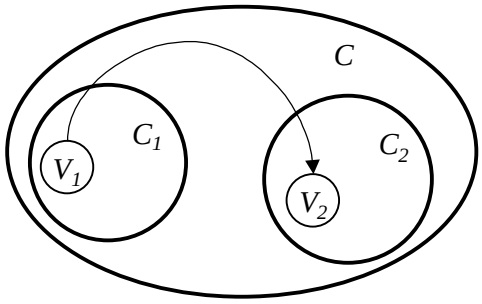


Figura 21: Nó de contexto de usuário *C* contendo dois nós e um elo

Portanto, o cancelamento automático de inscrições deve levar em consideração tais situações para que não haja a remoção indevida de inscrições.

Na ocorrência dos eventos descritos abaixo, determinadas inscrições realizadas pelo sistema serão automaticamente canceladas.

3.3.6.1 Remoção de um nó *N* de um contexto de usuário

Quando um nó *N* é removido de um nó de contexto de usuário *C*, todas as inscrições em *C* referentes a algum contexto de notificação de *N* e que foram incluídas pela regra de adição de um nó em um contexto de usuário devem ser canceladas.

3.3.6.2 Remoção de um elo de um nó de contexto de usuário

Seja *E* o elo removido de um nó de contexto de usuário *C* e V_i, $i \in [1,m]$, o conjunto de pontos terminais de *E*. Todas as inscrições referentes em *C* a algum contexto de notificação de V_i, $i \in [1,m]$, e que foram incluídas pela regra de adição de elo em um nó de contexto de usuário devem ser canceladas.

No caso de remoção de um ponto terminal a um elo *E* pertencente a um nó de contexto de usuário *C*, seja *V₁* o nó âncora do ponto terminal. A regras de remoção acima deve ser executada com *i* sempre igual a 1.

3.3.6.3 Estabilidade de um Nó

Quando um nó *V* recebe uma notificação de que um nó *V'* teve seu estado alterado de temporário para permanente, a inscrição em *V* referente a esse contexto de notificação de *V'* deve ser cancelada, já que esse evento ocorre somente uma vez em um nó *V*, sendo desnecessário manter tal inscrição.

3.3.6.4 Obsolescência de um Nó

Quando um nó *V* recebe uma notificação de que um nó *V'* teve seu estado alterado de permanente para obsoleto, as inscrições em *V* referentes a todos os contextos de notificação que *V'* publica notificações, exceto os contextos de notificação de destruição e de recebimento de notificação, devem ser canceladas. A preservação das inscrições em contextos de notificação que *V'*, mesmo que *V'* já esteja obsoleto, publica recebimento de notificações é útil, por exemplo, no caso em que *V* efetua repasse de notificação de eventos entre seus filhos de derivação e estes não estão obsoletos.

3.3.6.5 Destruição de um Nó

Antes de ser destruído, um nó *N* publica uma notificação no contexto de notificação (*N*, d*estruição*) informando os assinantes que todas inscrições referentes à *N* serão canceladas.

É importante considerar o cancelamento das inscrições relativas à eventos de recebimento de notificação. Isto é, um determinado nó N_1 não deve continuar inscrito em um nó N_2, evento de recebimento de notificação de um nó N_3 (seja ancestral, ascendente, descendente ou independente) que tenha sido destruído. Repare que o mecanismo de cancelamento removerá a inscrição de N_1 quando N_2 for destruído, mas não quando N_3 for destruído. É difícil capturar quais inscrições devem ser canceladas nesse caso, dadas as diversas possibilidades de inscrição. Portanto, algum mecanismo de coleta de lixo deve ser implementado para a remoção dessas inscrições.

4. Implementação

O objetivo principal deste capítulo é descrever a implementação do controle de versões, apresentado no Capítulo 2, e do mecanismo de notificação, proposto no Capítulo 3.

Java [JAVA 2000] foi a linguagem utilizada para a programação. Uma grande vantagem dessa linguagem é que ela proporciona execução de programas em ambientes de sistemas operacionais heterogêneos, além de ser orientada a objetos, facilitando a implementação das classes do NCM, um modelo orientado a objetos.

Primeiramente, o diagrama de classes desta implementação é mostrado na Seção 4.1. Em seguida, a implementação do mecanismo de versionamento é contemplada na Seção 4.2, continuando com a implementação do controle de notificação na Seção 4.3. Ainda neste capítulo, mais precisamente na Seção 4.4, serão mostrados alguns exemplos onde a notificação no versionamento de nós no NCM pode apoiar autores em ambientes de edição hipermídia, quando esses trabalham em documentos compartilhados.

Os códigos fonte das classes envolvidas nesta implementação estão disponíveis no Anexo I desta dissertação.

4.1 Diagrama de Classes

A Figura 22 mostra o diagrama de classes da implementação, que segue as convenções de notação da UML (*Unified Modeling Language*) [BOOCH 1996].

As classes que participam do controle de versões serão comentadas na Seção 4.2, e as classes que atuam no esquema de notificação serão abordadas na Seção 4.3.

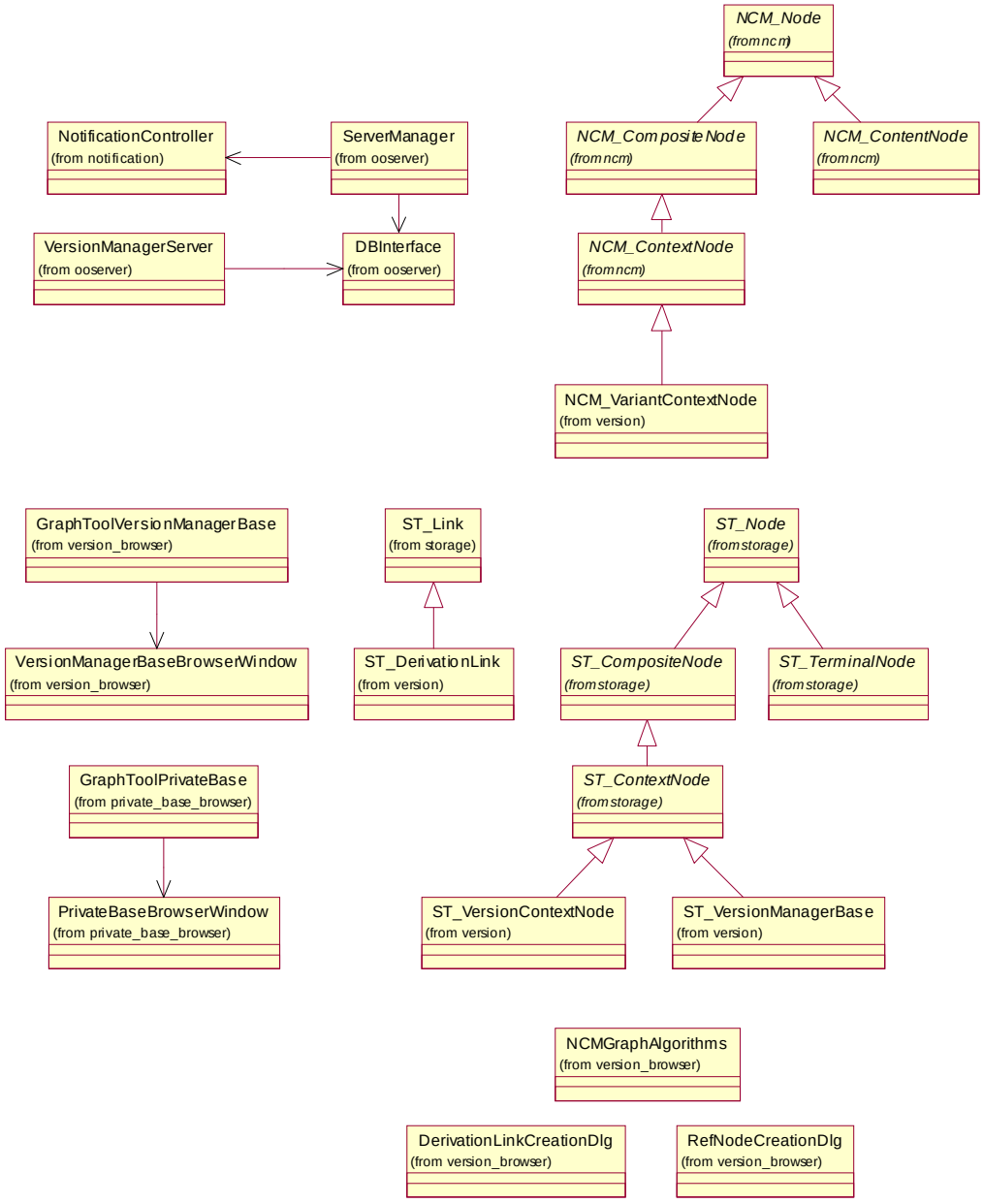


Figura 22: Diagrama de classes em UML

4.2 Controle de Versões

A implementação do controle de versões no HyperProp, a qual é constituída de módulos que são executados tanto no cliente como no servidor, foi caracterizada por diferentes fases.

Primeiramente, como o servidor HyperProp estava implementado sobre o banco de dados relacional Access [FERRET 1997], as estruturas de dados para controle de versões, como a base de contexto de versões e os contextos de versões, foram tornadas persistentes utilizando o mapeamento em tabelas desse banco de dados.

Uma outra característica dessa primeira implementação é que essas estruturas ficavam disponíveis em memória principal (RAM) do servidor. Na verdade, as operações de leitura e escrita dessas estruturas eram sempre realizadas em RAM, e uma operação de sincronismo com o banco de dados era efetuada a cada atualização de escrita. O objetivo dessa opção era tornar mais rápido o acesso às estruturas de versionamento, embora a RAM estivesse sendo consumida em demasia.

Em seguida, o servidor HyperProp foi migrado para utilizar o banco de dados orientado a objetos POET [POET 1998]. Essa migração possibilitou que os objetos das classes NCM, a serem tornados persistentes, fossem mapeados mais naturalmente no banco de dados.

No âmbito do controle de versões, o armazenamento em RAM das estruturas de dados para versionamento foi eliminado: as alterações são realizadas diretamente no POET. Para melhorar o desempenho de operações no banco de dados, o POET disponibiliza em RAM objetos mais acessados frequentemente, procedimento que realmente não deveria ficar a cargo do sistema hipermídia.

Embora não existam dados medidos precisamente, pode-se dizer, baseando-se em observações empíricas, que as operações de banco de dados no POET, relacionadas ao acesso às estruturas de versionamento, reveleram-se bem mais rápidas do que as mesmas implementadas sobre o Access.

A arquitetura atual da implementação do controle de versões é mostrada na Figura 23.

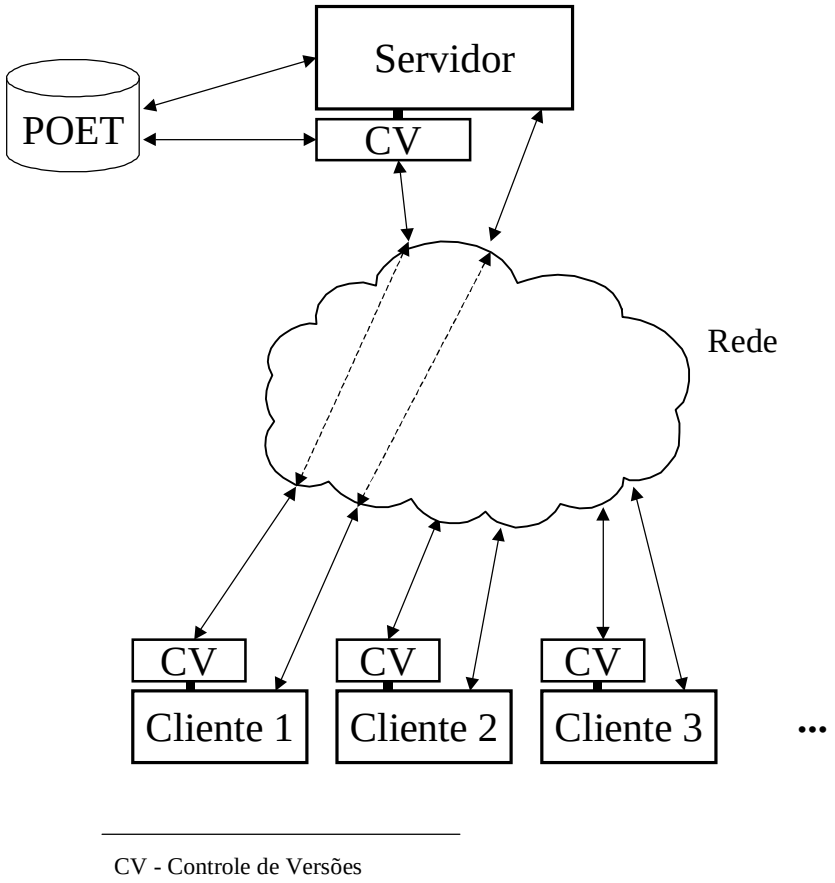


Figura 23: Arquitetura básica do controle de versões no sistema HyperProp

Nessa figura, os clientes e servidores estão separados por uma rede de comunicação de dados, contudo, uma máquina que executa o servidor pode executar também clientes, sem restrições.

O módulo de controle de versões do servidor, implementado pelas classes *VersionManagerServer* e *DBInterface*, e o servidor HyperProp, implementado principalmente pelas classes *ServerManager* e *DBInterface*, acessam diretamente o banco de dados POET:

- Módulo de controle de versões no servidor - para armazenar as entidades NCM relacionadas ao versionamento, por meio da classe *DBInterface*;
- Módulo servidor HyperProp - para armazenar as entidades NCM que não estão relacionadas ao versionamento, por meio também da classe *DBInterface*.

O módulo de controle de versões no cliente se comunica diretamente com o módulo de controle de versões no servidor para a execução de operações de versionamento, como o *check-out* e *check-in* de versões de dados. Um cliente pode se comunicar diretamente com o servidor para solicitar, por exemplo, um UID de uma nova base privada.

Para determinar que nós pertencem a quais bases, um nó *N* é identificado pelo par (UID_Base, UID_Local), onde UID_Base é o identificador único (UID) da base (hiperbase pública ou base privada) a que *N* pertence, atribuído pelo servidor, e UID_Local é o UID de *N* local a essa base, atribuído em cada cliente. A hiperbase pública recebe a identificação especial zero, e a cada base privada *BP* criada, o servidor dispensa um novo UID a *BP*. Por exemplo, um nó identificado por (5,1) pertence à base privada de identificação "5" e é o nó de identificação local "1" dessa base. Já um nó identificado por (0,7) pertence à hiperbase pública e é o nó de identificação local "7".

As classes *ST_Node*, *ST_Terminal*, ST_*CompositeNode*, *ST_ContextNode*, *ST_VersionContextNode*, *ST_VersionManagerBase* e *ST_Link* representam, respectivamente, as classes NCM nó, nó terminal, nó de composição, nó de contexto, nó de contexto de versões, nó base de contexto de versões e elo, que são utilizadas no servidor para o armazenamento no banco de dados. A classe *ST_DerivationLink* é utilizada para a persistência de elos simplificados que pertencem aos contextos de versões e à base de contexto de versões.

Já as classes *NCM_Node*, *NCM_ContentNode*, NCM_*CompositeNode*, *NCM_ContextNode* e *NCM_VariantContextNode* representam, respectivamente, as classes NCM nó, nó terminal, nó de composição, nó de contexto e nó de contexto de variantes, que são utilizadas para instanciar objetos em clientes.

No sistema HyperProp, cada usuário possui à disposição um *browser* de hiperbase pública [PINTO 2000], como mostrado na Figura 24. Para efetuar a operação de derivação de versões de dados, o usuário pode selecionar com o *mouse* o nó desejado, e depois escolher a opção "*check-out"* no menu "*operations*". Existem

outras formas de realizar o *check-out* por meio de interação na base privada, como pode ser visto na dissertação [PINTO 2000].

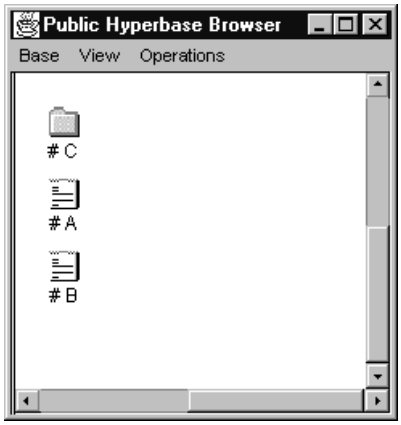


Figura 24: *Browser* de hiperbase pública - nós *A* e *B* (terminais), e *C* (ctx. de usuário)

Repare que, fisicamente, a hiperbase pública está armazenada no servidor, enquanto que cada cliente possui uma visão local dessa base, necessária para apresentar a estruturação do documento. Essa visão é formada pelos nós terminais e de contexto de usuário (incluindo elos) pertencentes à hiperbase pública, ou seja, todos os identificadores de nós (e de elos) são trazidos do servidor para o cliente, com o intuito de apresentá-los no *browser* de hiperbase pública.

Quando ocorre o *check-out* de um nó *V* gerando um nó objeto de dados *V'*, é verificado se existe algum nó de contexto de versões criado associado à *V*[18]*:*

- Caso não exista, um nó de contexto de versões *CV* é criado e inserido na base de contexto de versões. Uma referência *R* ao nó *V*, que contém dois atributos UID_Base e UID_Local relativos à *V*, é inserida em *CV*. Observe que um nó de contexto de versões, armazenado no servidor, pode conter nós que estão no próprio servidor, mas também nós que estão distruibuídos pelos clientes. Depois disso, o procedimento do caso abaixo, quando o contexto de versões já existe, é executado.

[18] Os nós terminais e de contexto de usuário possuem um atributo especial *VC_UID* que identificam o nó de contexto de versões associado.

- Caso exista o contexto de versões *CV* associado à *V*, uma referência *R'* ao nó *V'*, que contém dois atributos UID_Base e UID_Local relativos à *V'*, é inserida em *CV* . Seja *R* a referência ao nó *V*, um elo de derivação é criado ligando *R* a *R'*.

As alterações na base de contexto de versões e nos contextos de versões são realizadas diretamente na base de dados, por meio de métodos de acesso ao banco de dados implementados na classe *DBInterface*.

A operação de *check-out* de um nó da hiperbase pública, disparada por um cliente, traz a versão do nó requerida por meio da invocação remota de métodos (*Remote Method Invocation* – RMI), onde o atributo conteúdo de nós terminais é uma referência, em forma de URL (*Uniform Resource Locator*) [HETHMON 1997], ao conteúdo original do nó.

Para facilitar a identificação de versões derivadas de um nó, um esquema de numeração foi implementado. Quando um nó qualquer deriva uma versão, o sufixo "*.x*" é concatenado ao nome[19] do nó que derivou a versão, gerando um novo valor para o nome da versão derivada, onde *x* é igual a 1, se for a primeira versão derivada, 2 se for a segunda versão derivada, e assim por diante. Por exemplo, se um nó com nome igual a "PUC" derivar sua primeira versão, o nome desta versão será "PUC.1", e caso uma segunda versão seja derivada de "PUC", esta versão derivada receberá o nome de "PUC.2". Se a versão com nome igual a "PUC.1" derivar sua primeira versão, o nome desta versão será "PUC.1.1", e assim por diante.

No caso do *check-out* de um nó de contexto de usuário *C* na hiperbase pública que contém outro nó *N*, segundo o controle de versões do NCM, uma versão *C'* é criada e inserida na base privada em que a operação de *check-out* foi solicitada. *C'*, que está fisicamente em um cliente, deve conter *N*, que está fisicamente no servidor. Nesta implementação, o nó *C'* contém uma referência, que está fisicamente também

[19] Na implementação, a classe "entidade" do NCM possui o atributo nome.

nesse cliente, à *N*, mesmo porque podem existir elos visíveis em *N*, por exemplo, que devem ser mostrados no *browser* de base privada do cliente [PINTO 2000].

Essas referências em bases privadas a nós da hiperbase pública podem ser reconhecidas visualmente no próprio *browser* de base privada, como mostra a Figura 25. Nesse exemplo, o nó de contexto de usuário *C.1* contém os nós *A* e *B*, os quais pertencem à hiperbase pública (a "tralha" é a diferenciação).

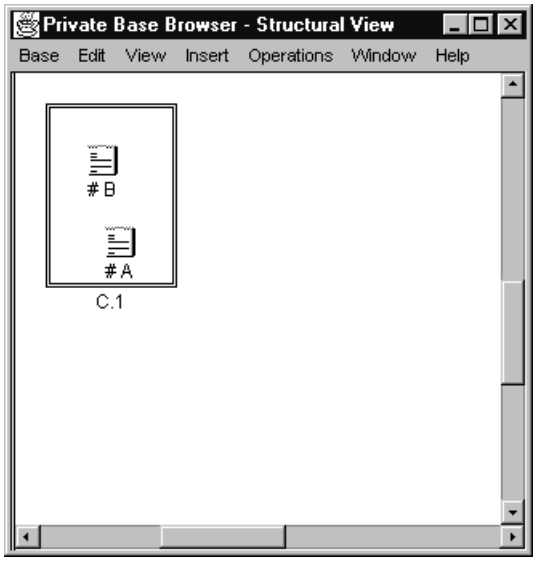


Figura 25: *Browser* de base privada

A operação composta *open*, para criação de versões de dados, foi implementada utilizando o próprio *check-out*. Alguns detalhes foram observados para a atribuição de novos UIDs, quando ainda não existisse versão derivada na base privada recursivamente contida em um nó de composição passado por parâmetro, e de reutilização de UIDs, quando já existisse uma versão na base privada também recursivamente contida no nó de composição passado por parâmetro. Essa atenção foi exigida especialmente no caso do *open*[20], pois esses UIDs precisariam ser trocados entre cliente e servidor para que nós criados (ou reutilizados) no *check-out* pudessem ser identificados pela primitiva composta *open*, que disparou a operação de *check-out*.

[20] Na verdade, todo esse suporte (parametrização, etc.) foi inserido na operação de *check-out*, cuja necessidade foi percebida mais facilmente na implementação do *open.*

É interessante observar que no caso do *open*, as operações de *check-out* ocorrem no sentido dos nós mais externos aos nós mais internos.

Para a operação de *check-in* de uma versão de dados de um nó de contexto de usuário, o comportamento é diferente do que ocorre no *open*: o *check-in* ocorre no sentido dos nós mais internos aos nós mais externos. Ou seja, um nó de contexto de usuário que sofre *check-in* deve primeiramente realizar o *check-in* de seus nós mais internos, recursivamente. Isso é importante para o banco de dados POET, que não pode armazenar um objeto que contém objetos, os quais não estão fisicamente armazenados, e sim somente as referências. Este seria o caso se o *check-in* enviasse primeiramente os nós mais externos.

Quando ocorre o *check-in* de um nó objeto de dados novo (não é versão de outro), ele simplesmente é movido para a hiperbase pública, nenhum contexto de versões ou base de contexto de versões é alterado.

No caso de uma versão de dados que não sofreu modificações em relação a versão original de que derivou, o nó é destruído. Quando a versão de dados é modificada, o nó é movido para a hiperbase pública e a referência no contexto de versões correspondente é atualizada de forma a apontar para o novo nó disponibilizado na hiperbase pública.

Quando um nó em uma base privada sofre a operação *delete*, passa para o estado obsoleto, caso esteja no estado permanente, ou então é diretamente destruído.

No caso da destruição de um nó *N*, seja por coleta de lixo ou pela remoção por ser um nó temporário, a referência que representa *N* no contexto de versões correspondente é removida. Os elos de derivação que tem como origem ou destino *N* são reorganizados da seguinte maneira:

- Para cada elo *E* que tem origem uma nó arbitrário *A* e destino *N*, *E* é replicado tendo como origem *A* e destino cada versão derivada de *N*, se houver. Caso *N* não seja versão de qualquer nó, os elos que partem de *N* são sumariamente removidos.

Quanto a mudança de estado, além dela ser efetuada pelo sistema automaticamente como efeito colateral de operações do modelo, o usuário pode também selecionar o nó na interface gráfica e escolher no menu de operações o novo estado do nó. Obviamente, restrições e comportamento do modelo para o estado de um determinado nó são aplicados.

Apesar de não existirem meios na interface gráfica do HyperProp para que o usuário solicite a edição em versões de representação, as primitivas de *check-out* e *check-in* de versões de representação, incluindo o comportamento de tais versões em mudanças de estado das versões, foram implementadas. De fato, por não haver, nesse caso, interação cliente-servidor, a implementação do controle de versões entre o plano de dados e o plano de representação foi menos complexo, porém mais trabalhoso devido aos diferentes comportamentos no tratamento de versões de representação, especialmente nas operações de mudança de estado, conforme comentado na Seção 2.3.4.2.

É bom ressaltar que, no caso de operações de derivação de versões de representação, os contextos de variantes não precisam ser tornados persistentes, pois existem somente em tempo de navegação ou edição.

Como comentado anteriormente, a base de contexto de versões, que pode conter elos de derivação para explicitar operações de fusão, fornece um histórico de derivação de documentos juntamente com os contextos de versões. É interessante que um usuário qualquer tenha acesso a esse tipo de informação, de preferência em uma interface gráfica de simples operação. Para isso, o *browser* de base de contexto de versões foi concebido. A classe que implementa as funcionalidades desse *browser* chama-se *VersionManagerBaseBrowserWindow.*

Esse *browser* herda as facilidades de visualização de documentos estruturados, incluindo mecanismos para filtragem como visão de olho-de-peixe [PINTO 2000], disponíveis no *browser* de base privada.

Um exemplo de um *browser* de base de contexto de versões é mostrado na Figura 26. Nesse exemplo, um nó *C* que pertence à hiperbase pública derivou o nó *C.1*

na base privada "*root*". Esses dois nós (referências) foram incluídos em um contexto de versões *VC_C*, que por sua vez, foi incluído na base de contexto de versões.

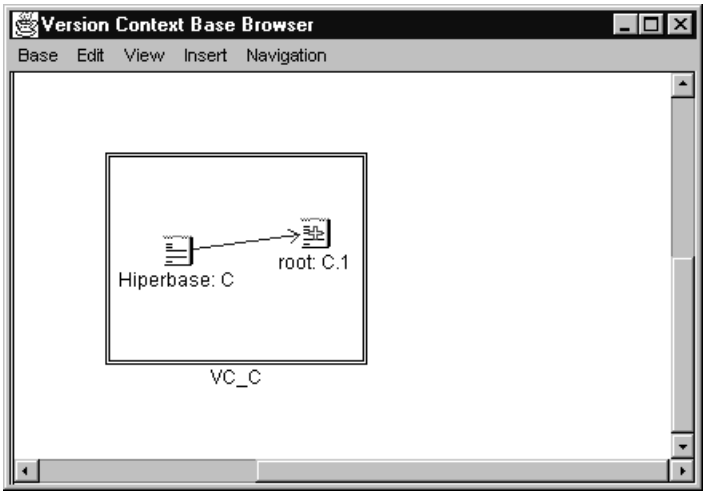


Figura 26: *Browser* de base de contexto de versões

Outro exemplo é apresentado na Figura 27, onde o contexto de versões de *V* está selecionado na área esquerda da janela por meio da visão em árvore [PINTO 2000], e o conteúdo do contexto de versões é mostrado na área direita da janela. Nesse exemplo, o nó *V* pertencente à hiperbase pública derivou as versões *V.1*, *V.2* e *V.3* nas bases privadas dos usuários *muniz*, *debora* e *arthur*, respectivamente.

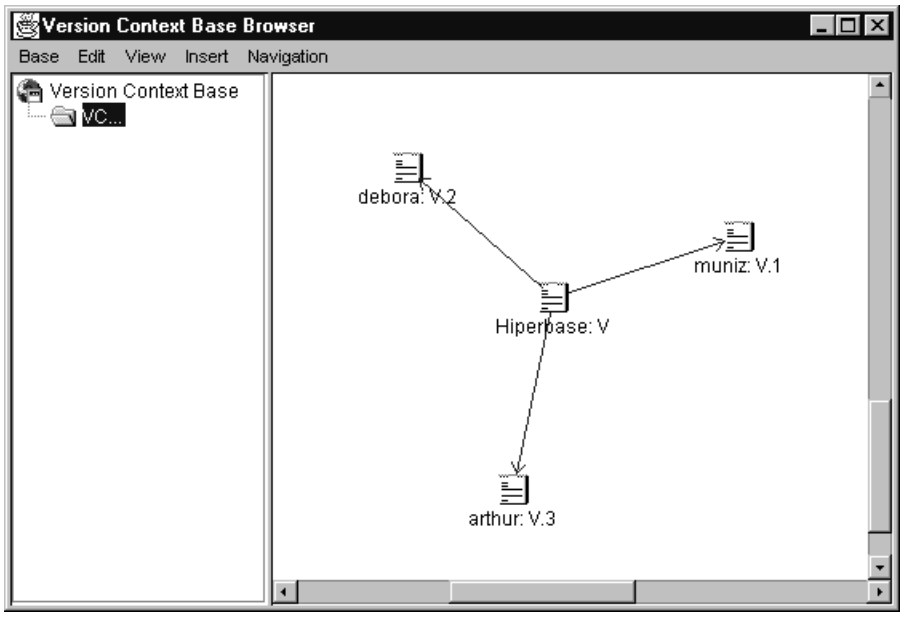


Figura 27: *Browser* de base de contexto de versões

O *browser* de base de contexto de versões oferece uma opção para inserção manual de elos de derivação. Essa inserção é implementada pela classe *DerivationLinkCreationDlg* e é analisada em um procedimento, implementado pela classe *NCMGraphAlgorithms*, para verificar se o grafo de derivação continuará acícliclo. A idéia geral do algoritmo implementado para detecção de ciclos no grafo de derivação pode ser encontrada no Apêndice A. O *browser* de base de contexto de versões também permite a inserção manual de nós (referências) em um nó de contexto de versões, por meio da classe *RefNodeCreationDlg*.

Nas primeiras implementações do controle de versões, o *check-in* de um nó objeto de dados realizado por algum usuário não atualizava a visão do *browser* de hiperbase pública dos clientes para apresentar o novo nó disponível. O procedimento de atualização dessa visão (*refresh*) era solicitado manualmente pelo usuário.

Na versão atual, o procedimento de atualização foi estendido de forma a capturar alterações na hiperbase pública (ou em algum de seus nós internos) realizadas por outros usuários. Para isso, o próprio mecanismo de notificação [Seção 4.3] foi usado para disparar notificações aos clientes. Dessa maneira, os clientes são informados que existe uma atualização na hiperbase pública, e solicitam a nova visão ao servidor.

Da mesma forma, o procedimento de atualização da visão da base de contexto de versões (também manual) não capturava mudanças em contextos de versões quando, por exemplo, usuários realizavam o *check-out* de um determinado nó da hiperbase pública (um novo nó é inserido no contexto de versões correspondente, juntamente com um elo que explicita a relação de derivação).

Atualmente, o procedimento de atualização da base de contexto de versões também foi estendido de maneira a informar, automaticamente, os clientes que uma visão (atualizada) dessa base está disponível no servidor.

4.3 Controle de Notificação

O mecanismo de notificação do NCM foi implementado e incorporado ao sistema HyperProp, utilizando como apoio o iBus. O *iBus* [IBUS 1998] é um *middleware* em Java usado para a implementação da comunicação baseada no paradigma *Publish/Subscribe* que segue uma API (*Aplication Program Interface*) padrão desenvolvida pela SUN™ chamada JMS (*Java Message Service*) [JMS 2000]. Ele fornece as funcionalidades básicas de criação e assinatura de contextos de notificação, publicação e recepção de mensagens, sempre tornando transparente a resolução que deve haver entre um contexto de notificação e seus assinantes: um nó, independente da base a que pertença, simplesmente se inscreve em um canal (contexto de notificação) para receber notificações.

Dado o estado da implementação atual, nenhum código teria de ser alterado, caso outro *middleware*[21] compatível com o padrão JMS substituísse o iBus. Para substituir o iBus por outras implementações que não sigam a API JMS, mas que ofereçam as funcionalidades do *Publish/Subscribe*, como o *Corba Notification Service* [CORBA 1998], seriam necessárias alterações no código referentes basicamente à sintaxe de funções.

A arquitetura básica proposta do controle de notificação no sistema HyperProp é mostrada na figura seguinte:

[21] Atualmente existem cerca de 10 implementações do padrão JMS [JMS 2000].

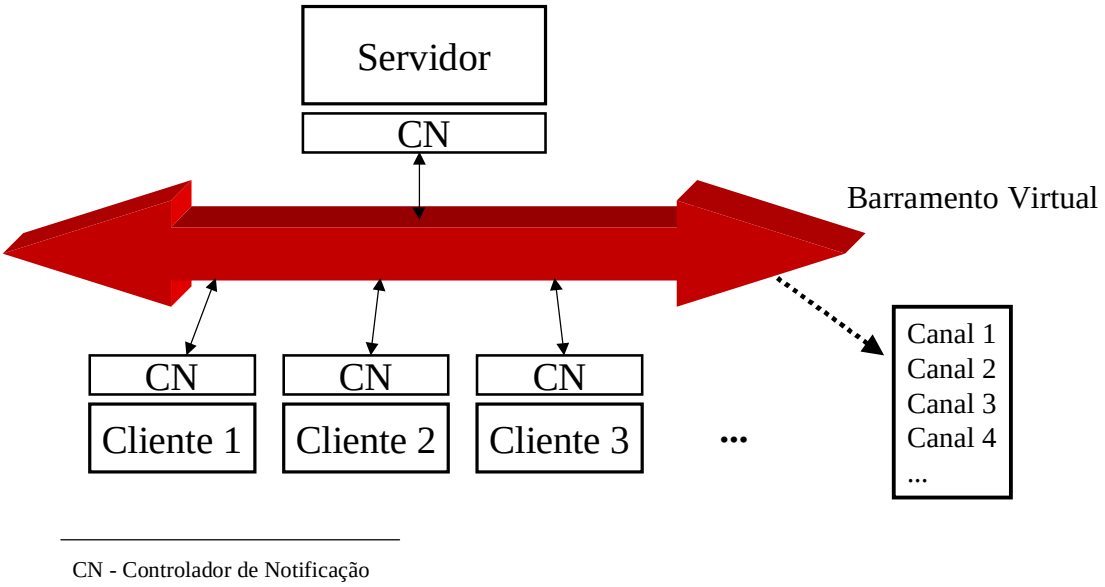


Figura 28: Arquitetura básica do controle de notificação no sistema HyperProp

Cada cliente ou servidor instancia um controlador de notificação (CN), que é um objeto da classe *NotificationController*. O CN é responsável pelas assinaturas e cancelamentos de canais, publicação e recepção de notificações de todos os nós pertencentes a cada cliente ou servidor, e utiliza diretamente as funcionalidades oferecidas pelo iBus.

O conceito de nós de contexto de notificação foi implementado de uma maneira diferente. Em vez dos nós de contexto de notificação possuírem as informações dos nós que contém, um nó qualquer (inclusive um nó *container* de notificação, exceto nós de contexto de notificação) mantém a informação dos contextos de notificação, que são canais bem conhecidos [Apêndice B], em que está inscrito. Para isso, o atributo lista de contextos de notificação foi adicionada às classes *NCM_Node* e *ST_Node*.

O atributo *lista de contextos de notificação* de um nó *N* identifica em quais contextos de notificação *N* está inscrito para receber notificações. Por exemplo, seja *L* a lista de contextos de notificação de um nó *N*:

$$L = \{(N_b, \textit{obsolescência}), (N_e, \textit{derivação de versão})\}$$

Essa lista, que contém dois contextos de notificação, indica que o nó *N* deve receber notificações da obsolescência de N_b e da derivação de versão de N_e.

O atributo lista de contextos de notificação permitirá a recepção de notificações por um nó. Para a transmissão de mensagens de notificação, nenhum atributo extra é necessário. Simplesmente um nó publica a notificação no canal (contexto de notificação) correspondente à sua identificação e ao evento de disparo.

É claro que, como comentado anteriormente, deve existir um resolução entre um contexto de notificação e seus assinantes. No entanto, essa resolução *Contexto de Notificação↔Assinantes* é uma abstração fornecida pelas facilidades do paradigma *Publish/Subscribe*, disponibilizadas pelo iBus. Na implementação, uma grande vantagem de atribuir essa tarefa de resolução ao mecanismo de comunicação é que um assinante pode se inscrever ou sair de um canal sem ter de se preocupar com os editores e vice-versa.

Dois métodos foram adicionados à classe *NCM_Node*:

1. *Assina contexto de notificação*: insere um contexto de notificação na lista *L* de contextos de notificação de *N*. Recebe como parâmetro o identificador de um nó *N'* e o evento de disparo *E*, os quais formam um contexto de notificação. Caso não exista qualquer contexto de notificação referente à *N'* em *L*, o contexto de notificação especial (*N'*, *Destruição*) é incluído automaticamente em *L*. Além disso, um registro junto ao mecanismo responsável pela resolução *Contexto de Notificação↔Assinantes* é realizado, ou seja, métodos de inscrição em canais oferecidos pelo iBus são invocados.

2. *Cancela contexto de notificação*: remove um contexto de notificação da lista de contextos de notificação de *N*. Ao remover um contexto de notificação (*N'*, *E*), é verificado se ainda existem outros contextos de notificação referentes à *N'*. Caso não exista, o contexto de notificação especial (*N'*, *Destruição*) é removido. Além disso, a remoção do registro junto ao mecanismo responsável pela resolução *Contexto de Notificação↔Assinantes* é realizada, ou seja, métodos de cancelamento em canais oferecidos pelo iBus são invocados

Na implementação do modelo de notificação, um nó N_1 não se inscreve diretamente em outro nó N_2, as bases desses nós funcionam como procuradoras. Ou seja, a base a que N_1 pertence se inscreve na base a que N_2 pertence para receber notificações, e as bases repassam as notificações aos nós correspondentes.

A cada evento de disparo que ocorre em um determinado nó, uma mensagem de notificação é publicada, via iBus, no canal associado a esse nó e ao evento de disparo. O iBus, por sua vez, executa a resolução *Canal↔Assinantes*, determinando quais bases devem receber a notificação. Após a publicação da notificação, as regras de inscrição correspondentes ao evento de disparo (se existirem) são processadas para a geração de novas inscrições.

Como o nó contém a lista de contextos de notificação em que está inscrito, a persistência dessa lista pode ser obtida com a implementação da persistência do próprio nó, que é o caso de nós na hiperbase pública (banco de dados) e em nós de base privada (armazenamento pelo próprio sistema HyperProp). As alterações na lista de contextos de notificação em nós pertencentes à hiperbase pública são realizadas diretamente no POET, semelhante ao que ocorre nas modificações em contextos de versões e na base de contexto de versões.

Para que clientes possam ser informados que devem solicitar uma nova visão (atualizada) da hiperbase pública ou da base de contexto de versões, como comentado na Seção 4.2, foram criados dois contextos de notificação (canais) especiais no servidor:

- "*PublicHyperbaseChanged*"
- "*VersionContextBaseChanged*"

O primeiro está associado à mensagens de alteração na hiperbase, enquanto que o segundo, na base de contexto de versões (ou em contexto de versões). Desse modo, os *browsers* de hiperbase pública e de base de contexto de versões em cada cliente se inscrevem nos canais "*PublicHyperbaseChanged*" e "*VersionContextBaseChanged*", respectivamente, ao serem instanciados inicialmente. Ao receberem mensagens de notificação nesses canais, os *browsers* solicitam ao servidor a visão atualizada da base correspondente.

No sistema HyperProp, uma janela, instanciada no servidor e em cada cliente, apresenta mensagens de notificação e também os usuários que estão conectados ao sistema. À medida em que um determinado usuário efetua a conexão ou desconexão no sistema, ou recebe notificações (por meio do sistema), as janelas (servidor e clientes) são atualizadas dinamicamente.

Essa janela, denominada no sistema de *janela de notificação*, é mostrada na Figura 29. Os usuários *root*, *muniz*, *arthur* e *debora* estão conectados no sistema através dos endereços de rede (*IP*) 139.82.95.14, 139.82.95.13, 139.82.95.85 e 139.82.95.28, respectivamente. O espaço vazio no lado direito é o local onde são impressas as notificações que são recebidas pelos nós da base privada do usuário.

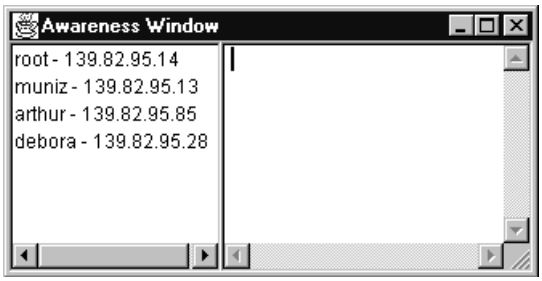


Figura 29: Janela no HyperProp exibindo usuários e notificações do sistema

Para implementação da lista de usuários conectados em um dado momento, o servidor HyperProp foi acrescido de uma lista de todos os usuários conectados ao sistema, onde cada cliente é inscrito automaticamente[22], por meio do iBus, no canal especial "*LoggedUsersChanged*". Quando algum usuário entra ou sai do sistema, é publicada

[22] Na verdade, é a OD-base privada que representa o cliente na inscrição.

uma mensagem nesse canal especial, informando que a lista de usuários do sistema foi alterada. Então, cada cliente solicita ao servidor a lista de usuários atualizada.

Existe ainda uma opção na interface gráfica para que o usuário, após a seleção de um nó, possa solicitar a inscrição manual em um canal.

4.4 Estratégias Reativas

Como mencionado anteriormente, o controle de notificação atua no NCM propagando informações sobre eventos de edição que ocorrem em nós. A princípio, as mensagens enviadas pelo esquema de notificação possuem caráter informativo, cabendo ao sistema de autoria absorver a informação da notificação e tomar as providências adequadas.

Em sua implementação atual, o sistema HyperProp apresenta as notificações recebidas por um nó *N* pertencente a uma base de documentos *B* (hiperbase pública ou base privada), em uma janela associada a *B*, disponível na interface de cada usuário. Isto é, uma janela é apresentada no servidor HyperProp, exibindo as notificações enviadas a nós da hiperbase pública, e também em cada cliente *C*, mostrando as notificações enviadas a nós pertencentes à base privada de *C*. Com isso, o usuário toma conhecimento do fato notificado e pode executar o procedimento que achar conveniente.

Essas notificações, no entanto, podem extrapolar a característica informativa e promover mecanismos reativos automáticos no sistema de autoria, úteis para a edição cooperativa.

O objetivo desta seção é mostrar algumas estratégias reativas, que provocam ações no sistema de autoria no recebimento de notificações. Essas estratégias, embora não implementadas no sistema HyperProp, mostram que o controle de notificação proposto pode ser utilizado como base para diferentes procedimentos relacionados à edição cooperativa.

Antes de descrever as técnicas propriamente ditas, é importante lembrar o que acontece quando usuários desejam acesso simultâneo a um nó disponível na hiperbase pública.

Suponha uma hiperbase pública H_B que contém um nó *V*, formado somente por atributos versionáveis. Suponha também os usuários *X*, *Y* e *Z*, em suas respectivas bases privadas, utilizando o sistema simultaneamente. Quando algum usuário quer editar ou visualizar *V*, ele realiza um *check-out* de *V* e começa a trabalhar na versão de *V* em sua base privada. Após concluir o trabalho, ele efetua o *check-in* da versão de *V*, devolvendo-a para a hiperbase pública. Notificações são propagadas comunicando o acontecimentos desses fatos aos demais usuários (por meio de nós).

É bom observar que as técnicas descritas a seguir podem ser combinadas entre si, de modo a atingir um resultado ainda mais produtivo.

4.4.1 Comunicação na Edição Semi-Síncrona

A primeira estratégia está relacionada à capacidade que o sistema de autoria tem de capturar o início de uma autoria semi-síncrona.

Segundo o controle de notificação, se o usuário *X* possui uma versão de *V* em sua base privada e o usuário *Y* desejar trabalhar em *V*, *Y* efetua um *check-out* de *V*, criando uma nova versão de *V* na base privada de *Y*. Uma notificação automática será enviada à versão de *V* contida na base privada de *X* (e exibida na *awareness window* de *X*), comunicando o início de uma autoria semi-síncrona.

Nesse caso, o sistema de autoria poderá, automaticamente, estabelecer um canal de comunicação (áudio ou texto) entre *X* e *Y*, no intuito de possibilitar que os dois troquem informações para coordenar as ações a serem realizadas. Ao final da edição, cada um pode executar o *check-in* da versão de *V*, gerando duas novas versões e rompendo o canal de comunicação estabelecido previamente.

4.4.2 Disparo Automático da Operação de Fusão de Nós

A segunda estratégia tenta disparar a execução de algoritmos para fusão de nós em momentos considerados relevantes para tal operação.

De acordo com o esquema de notificação, se o usuário *X* possui uma versão de *V* em sua base privada, denominada *V'*, e outro usuário efetua o *check-in* de outra versão de *V*, denominada *V''*, *X* é informado que existe uma versão mais nova de *V*.

Caso *V'* tenha sido modificada, o sistema de autoria pode oferecer, automaticamente, um algoritmo de fusão (interativo ou não) para que as alterações efetuadas em *V'* e as efetuadas em *V''* sejam mescladas, gerando uma nova versão temporária na base privada de *X*. É bom ressaltar que o *log* de notificações de *V'* e *V''* pode ser consultado para auxiliar a operação de mesclagem das versões.

Caso contrário, o sistema de autoria pode substituir automaticamente *V'*, desde que *V'* esteja no estado temporário, pela versão recém criada *V''*. Critérios podem ser adotados nessa substituição automática como, por exemplo, somente realizar a substituição quando os autores de *V* e *V''* forem os mesmos.

4.4.3 Apoio à Sincronização de Versões

A terceira estratégia tem como objetivo apoiar a sincronização na edição de atributos não versionáveis, quando essa ocorre em versões irmãs no grafo de derivação.

Suponha agora que o atributo conteúdo de *V* seja não versionável e que *X*, *Y* e *Z* possuem, respectivamente, *V'*, *V''* e *V'''* em suas bases privadas.

O comportamento padrão do modelo de notificação é informar as atualizações em atributos não versionáveis que ocorrem entre vizinhos no grafo de derivação. O sistema de autoria pode tentar sincronizar os conteúdos de *V'*, *V''* e *V'''*, transportando informações de atualização do atributo conteúdo (não versionável) na mensagem de notificação, a qual possui uma lista de atributos genéricos, como visto na Seção 3.3.4.

Observe que algum controle de concorrência distribuído se faz necessário para sincronizar e coordenar as ações concorrentes efetuadas por *X*, *Y* e *Z*. Esse controle

deve lidar com usuários que podem estar separados por redes de comunicação geograficamente distribuídas e de latência de propagação não-determinística.

5. Trabalhos Relacionados

O objetivo deste capítulo é apresentar e discutir alguns sistemas hipermídia relacionados a versões e notificação para o suporte à autoria cooperativa, sempre comparando-os com o trabalho desenvolvido ao longo desta dissertação.

Para avaliação dos trabalhos abaixo, uma questão importante foi considerada: o impacto do funcionamento do mecanismo de notificação no caso de versionamento de objetos hipermídia.

5.1 HyperBase e EHTS

O HyperBase [WIIL 1993] é um sistema de banco de dados hipertexto (*hypertext database*) desenvolvido na Universidade de Aalborg[23], Dinamarca, que oferece funcionalidades para aplicações hipermídia. O sistema captura a idéia geral de hipertexto, provendo operações em entidades básicas como nós e elos.

O sistema, implementado em C++, é baseado no modelo cliente-servidor. A arquitetura do servidor, ilustrada na Figura 30, é formada por três camadas: entidades básicas, serviços básicos e serviços multiusuário.

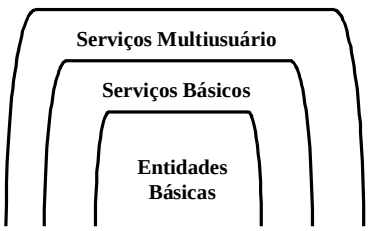


Figura 30: As diferentes camadas do servidor HyperBase

[23] Na realidade, existem dois sistemas com o mesmo nome "HyperBase". Um que foi desenvolvido na Dinamarca, assunto dessa seção, e outro, desenvolvido na Alemanha, que deu origem a outros sistemas hipermídia como o SEPIA [Seção 5.4].

A camada *entidades básicas* opera diretamente sobre o sistema de arquivos do sistema operacional, executando as tarefas de armazenamento e recuperação de nós e elos.

A camada intermediária, chamada de *serviços básicos*, executa operações em entidades, como criação e deleção de nós e elos, modificação de atributos de nós, além de prover primitivas de versionamento.

O controle de versões do HyperBase oferece funções para criação, deleção e manipulação de versões de nós[24], mas não de elos. Esse mecanismo de versões, inspirado no RCS (*Revision Control System*) [TICHY 1985], foi estendido para incluir as noções de versões globais, que são objetos visíveis para todos os usuários do sistema; e versões locais, que permitem usuários trabalharem em objetos de maneira privativa.

A última camada, denominada de *serviços multiusuário*, é responsável pelo suporte à autoria cooperativa. É nesse nível que o controle de notificação de eventos, dentre outros mecanismos, atua.

O mecanismo de notificação oferece aos usuários a possibilidade de serem informados quando outros usuários executarem ações importantes no banco de dados, ou quando certos eventos críticos que lhes interessam ocorrerem em objetos (nós ou elos), apesar de não existir a noção de grupo de usuários que trocam notificações privativas entre si.

Os usuários podem se inscrever[25] em operações ocorridas em qualquer atributo, tanto de nós, como de elos. Assim, o evento é identificado pela tripla (objeto, operação, atributo). O campo *operação* pode assumir, dentre outros, os valores: criação_nó, criação_elo, obtenção_trava (*lock*), liberação_trava (*unlock*), remoção_nó,

---

[24] Os nós no HyperBase são sempre de conteúdo, não existindo o conceito de nós de composição. Aplicações que necessitarem de nós da última classe, têm de implementar o próprio tratamento.

[25] As inscrições não são persistentes [WIIL 1993a].

escrita. Por exemplo, um usuário *X* tem à sua disposição inscrições em eventos do tipo:

- Evento(todos, escrita, conteúdo): *X* é informado quando o atributo conteúdo (*data attribute*) de qualquer nó (somente nós possuem o atributo conteúdo) sofre uma operação de escrita.

- Evento(todos, obtenção_trava, todos): *X* é informado quando qualquer atributo de qualquer objeto for travado (*locked*).

- Evento(todos, todas, todos): *X* é informado de todas as operações em todos atributos de todos os objetos, útil para um registro do sistema (*system logger*), por exemplo.

Cabe ao sistema que irá atuar sobre o HyperBase utilizar esse controle de notificação para efetuar as inscrições desejadas. O HyperBase não gera notificações automáticas para situações peculiares do sistema, como operações de versionamento. Para isso, deve existir um mecanismo que saiba capturar estados do sistema para disparar notificações automáticas.

O HyperBase já serviu de base para alguns sistemas hipertexto cooperativos como, por exemplo, o EHTS (Emacs HyperText System) [WIIL 1993b]. O EHTS suporta edições de arquivos textos realizadas por usuários utilizando o sistema simultaneamente, e aplica as funcionalidades de notificação de eventos disponíveis pelo HyperBase. Por exemplo, no EHTS, os usuários são providos de quatro tipos de notificação:

1. Intenção – Todos usuários são notificados quando um usuário *X* sinaliza a intenção de modificar o atributo conteúdo de um nó, por meio da obtenção de uma trava. Os usuários, que percebem a intenção e a identidade de *X*, podem então se inscrever no evento de liberação dessa trava[26].

[26] Os usuários são questionados se desejam se inscrever no evento de liberação da trava.

2. Atualização – Quando um usuário modifica o conteúdo de um nó, todos os usuários recebem o novo nó com o conteúdo atualizado, o qual é apresentado na interface gráfica do usuário, refletindo as alterações efetuadas.

3. Conclusão – Quando um usuário termina a modificação no conteúdo de um nó *N*, usuários que estão inscritos nesse evento são notificados, indicando que *N* está disponível para escrita.

4. Deleção – Quando um nó *N* é deletado, *N* é removido da interface gráfica de cada usuário.

5.2 Hyperform

O sistema Hyperform [WIIL 1997], continuação do HyperBase dinamarquês apresentado na Seção 5.1, tem como objetivo principal reestruturar as facilidades oferecidas pelo HyperBase, de modo a torná-lo mais extensível. A idéia é prover suporte hipertexto à aplicações, proporcionando extensibilidade através de especificações definidas em uma linguagem de programação[27].

O sistema Hyperform, ilustrado na Figura 31, é constituído por três componentes:

⇒ HBMS (Hyperbase Management System)

⇒ Integrador de Ferramentas (*Tool Integrator* – TI)

⇒ Editores

[27] No caso, o Hyperform utiliza a linguagem Scheme [SPRINGER 1990], um dialeto do Lisp [WINSTON 1988]. É bom observar que essa linguagem foi disponibilizada para definições de extensões ao Hyperform, e utilizada para a implementação do próprio sistema, apesar de módulos críticos terem sido implementados em C por questões de desempenho.

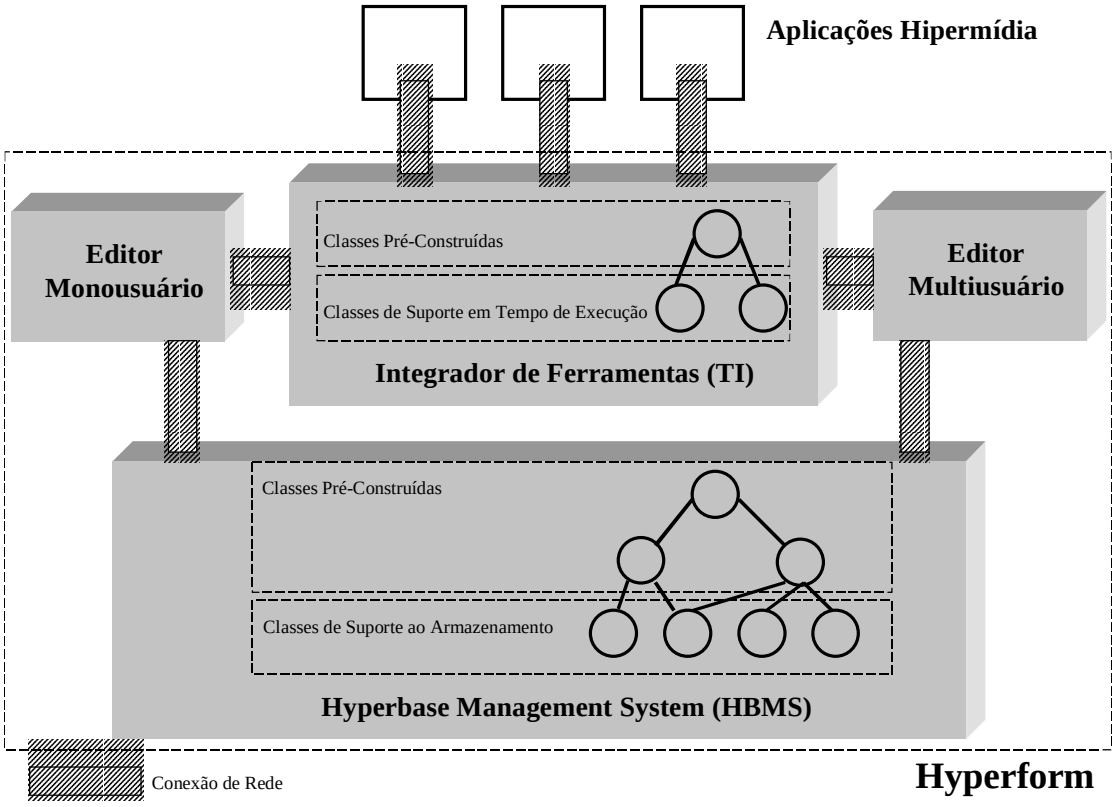


Figura 31: Componentes do sistema Hyperform

O TI age como uma interface entre o HBMS, o qual implementa o armazenamento de objetos, e as aplicações hipermídia. O suporte em tempo de execução para as aplicações é implementado nas classes de suporte em tempo de execução no TI, e o suporte ao armazenamento é implementado nas classes de suporte ao armazenamento do HMBS.

Novos serviços hipermídia são criados nos editores pela especialização e extensão das classes pré-construídas do HBMS e do TI. Essas classes, especializadas ou estendidas, geram classes de suporte ao armazenamento do HBMS, e classes de suporte em tempo de execução do TI.

Classes de suporte ao armazenamento do HBMS podem incluir âncoras, elos, nós de conteúdo e composição, herdando das classes pré-construídas, as características desejadas. Dentre as classes pré-construídas, existem aquelas que oferecem suporte ao controle de notificação e ao controle de versões, chamadas de *notification control* (NC) e *version control* (VC), respectivamente.

O controle de notificação, baseado nas experiências com o HyperBase dinamarquês, introduz uma idéia interessante para inscrição em eventos: as inscrições, que atualmente [WIIL 1997a] são persistentes[28] e armazenadas em uma lista interna do sistema, são especificadas utilizando a linguagem Scheme. Cada elemento dessa lista contém informações sobre o usuário inscrito e a expressão em Scheme usada para disparar eventos. Novamente, não existe a noção de grupo de usuários que trocam notificações privativas entre si.

Essas expressões permitem o acesso a todas as variáveis e funções do Hyperform, proporcionando assim, um mecanismo de notificação de eventos bem mais expressivo do que o HyperBase dinamarquês.

A Figura 32 mostra um exemplo de inscrição, em que operações de escrita (*write*) em atributos de dados (*data*) de qualquer objeto irão disparar um evento especificando o objeto destino e o usuário que realizou a operação (NC-entity, user).

```
(event(lambda ()
      (if (and (equal? NC-attribute “data”)
               (equal? NC-operation “Write”))
          (list NC-entity user)
          #f)))
```

Figura 32: Exemplo de uma inscrição usando a linguagem Scheme

Um outro exemplo de inscrição é apresentado na Figura 33. Todas as operações no objeto 5 realizadas pelo usuário “X” irão disparar um evento especificando o tipo da operação (NC-operation).

```
(event(lambda ()
      (if (and (= NC-entity 5)
               (equal? user “X”))
          (NC-operation)
          #f)))
```

Figura 33: Uma inscrição em operações no objeto 5 realizadas pelo usuário “X”

[28] Na verdade, as primeiras versões do Hyperform [WIIL 1992] não forneciam persistência para inscrições em eventos, característica que foi adicionada em trabalhos posteriores.

As expressões de cada inscrição são avaliadas quando o método "*send-events*" é acionado, o que acontece sempre que há uma atualização de atributos, aquisição ou liberação de travas, etc. Todas as expressões avaliadas que atingem um resultado diferente de falso geram um evento que será transmitido ao usuário inscrito.

Esse mecanismo de eventos pode ser utilizado para tornar o servidor HBMS ativo, onde operações específicas são capazes de disparar outras operações, as quais, por exemplo, podem atualizar variáveis e objetos no servidor HBMS automaticamente.

O controle de versões é bastante simples e é inspirado no sistema RCS [TICHY 1985], onde um determinado objeto tem sua árvore[29] de versões (histórico) armazenada. Versões são manipuladas por meio de operações de *check-out* e *check-in*, podendo ser armazenadas como completas (complete version) ou na forma de deltas (delta version). Nesse último caso, o armazenamento de uma versão *V'* derivada de *V*, implica armazenar as diferenças de *V'* em relação a *V*.

Esse mecanismo atua no versionamento de um objeto simples, não considerando versionamento, por exemplo, de objetos compostos, como nós de composição. Também não existe interação automática com o esquema de notificação, no sentido em que novas versões criadas pudessem notificar versões irmãs no grafo de derivação, dentre outras facilidades.

Alguns sistemas foram desenvolvidos utilizando o Hyperform, como por exemplo, o sistema HyperDisco (Hypermedia platform for distributed collaborative computing environments) [WIIL 1997], também desenvolvido na Universidade de Aalborg, Dinamarca.

O objetivo principal desse sistema é permitir que ferramentas distribuídas e heterogêneas de usuários utilizem, de maneira integrada, facilidades hipermídia como criação de elos entre objetos de ferramentas distintas, além de outras funcionalidades como armazenamento e controle de versões. Na verdade, as funcionalidades disponíveis no HyperDisco para notificação e controle de versões são basicamente as

[29] Existem derivações 1:N, mas não N:1.

mesmas oferecidas pelo Hyperform. Nenhuma extensão para aumentar o escopo de atuação de tais mecanismos foi realizada.

5.3 DeVise Hypermedia

[GRONBAEK 1994] descreve a implementação[30] de um sistema hipermídia, chamado DeVise Hypermedia (DHM). O sistema oferece suporte à autoria cooperativa e é baseado no modelo de referência hipertexto Dexter [HALASZ 1994], utilizando, portanto, conceitos como componentes, nós atômicos (terminais), nós de composição, âncoras, elos[31], etc.

A arquitetura do DHM é formada três camadas: *camada de armazenamento* (*Storage Layer*), *camada tempo de execução* (*run-time layer*) e *camada interior de componente* (*within-component layer*).

Na camada de armazenamento, atua um servidor de banco de dados orientado a objetos (BDOO), que foi desenvolvido com o intuito de oferecer facilidades para armazenamento de objetos hipermídia, considerando as necessidades de um sistema cooperativo, incluindo o compartilhamento simultâneo desses objetos por diversos usuários.

Na camada tempo de execução, existe o RP (*run-time process*), que provê serviços hipermídia para um conjunto de editores (camada interior de componente) utilizados por um usuário[32]. O RP é um cliente do servidor BDOO, servidor para os editores, e oferece as funcionalidades semelhantes ao integrador de ferramentas (*tool integrator*) do sistema HyperForm, sistema que será comentado nesse capítulo. Os RPs são também responsáveis pela distribuição de mensagens de notificação de eventos, enviadas pelo servidor BDOO, aos editores.

---

[30] Disponível para as plataformas UNIX e Macintosh.

[31] No modelo Dexter, nós atômicos, de composição e elos são denominados de componentes, os quais formam hipertextos.

[32] Existe um RP ativo para cada usuário ativo do sistema.

Na camada interior de componente, atuam os editores para usuário final, como por exemplo, editores gráficos, de texto, etc. Um editor efetua a manipulação, em tempo de execução, do conteúdo de um tipo de componente, onde a comunicação com objetos de armazenamento é sempre realizada por meio de um RP.

O compartilhamento de objetos é suportado por meio de mecanismos de travas (*locks*) e de notificação, os quais são fornecidos pelo BDOO. O mecanismo de notificação do sistema permite inscrições em eventos ocorridos em objetos compartilhados, podendo ser realizadas manualmente pelos usuários, ou então efetuadas automaticamente por uma aplicação hipermídia específica. Na verdade, o DHM não fornece qualquer mecanismo padrão para inscrições automáticas, apesar de tal funcionalidade poder ser obtida por meio de uma aplicação específica que utilizará o sistema. Também não existe qualquer mecanismo para controle de versões no DHM[33], reduzindo a quantidade de eventos relevantes para notificação tratada pelo sistema.

Para o suporte ao controle de notificação, o servidor BDOO do sistema é capaz de informar os seus clientes a ocorrência de vários tipos de eventos nos objetos armazenados, como por exemplo:

⇒ Criação, deleção e atualização de documentos hipertexto inteiros;

⇒ Criação, deleção e atualização de componentes (nós atômicos, elos ou composição) em um hipertexto;

⇒ Criação, deleção e atualização de âncoras e atributos em um componente;

⇒ Obtenção ou liberação de uma trava de escrita (*write lock*) em componentes e em hipertextos.

Assim, uma inscrição, que um cliente realiza para receber notificações, é identificada por um tipo de evento, um objeto (ou uma classe inteira de objetos), e uma

[33] E nem em seus sucessores, os quais serão comentados abaixo, apesar desse controle ser mencionado como trabalho futuro.

especificação de grupo de usuários. Por exemplo, um usuário pode se inscrever na atualização de um componente, quando essa atualização for feita por um dos integrantes de um dado grupo de usuários. É bom observar que um grupo de usuários pode ser formado por um único usuário, vários, ou por todos.

Uma opção interessante disponível na inscrição de eventos de atualização é a possibilidade de atualização imediata (*immediate update*). Dessa maneira, quando um componente recebe uma notificação de atualização e a inscrição correspondente a essa notificação contiver a opção de atualização automática ativada, o sistema automaticamente realiza a busca do componente atualizado (*refetch*).

A Figura 34 ilustra a interface do usuário disponível para operações em componentes, incluindo operações de inscrição. Na figura, um pequeno ícone de um sino aparece ao lado do componente "bp2159", indicando que o referido componente recebeu uma notificação. Observe a opção da atualização manual do componente (selecionado também) "bp2159" por meio da ação "*refetch*" no menu "*components*".

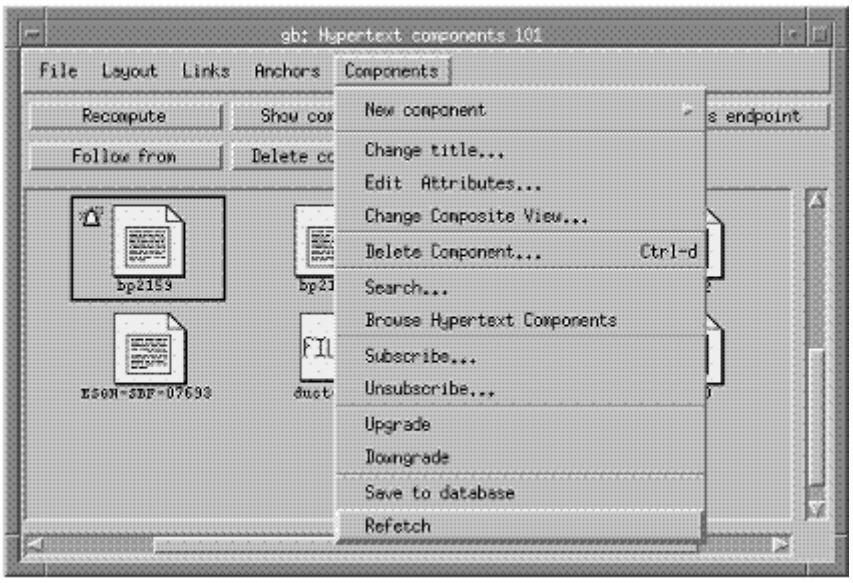


Figura 34: Interface do usuário no DHM

Cronologicamente, o DHM deu origem aos sistemas comerciais (HyperVise e WebVise) [GRONBAEK 1999]. O HyperVise é uma versão do DHM de implementação mais estável e robusta, que adiciona facilidades hipermídia disponíveis no DHM à aplicações existentes bem conhecidas, como o editor de texto Word™. O

WebVise é basicamente o mesmo sistema, porém é voltado para a integração com a WWW (World Wide Web – Web). Com origem também em meios acadêmicos [GRONBAEK 1997], o WebVise tem como objetivo expandir as funcionalidades hipermídia existentes em documentos na Web, como, por exemplo, a criação de elos em páginas em que não se possui direito de acesso para escrita. Um fato interessante do WebVise, além dos próprios mecanismos de suporte à autoria cooperativa oriundos do DHM, é que existe um esforço para utilização de mecanismos disponíveis em servidores WebDAV[34], como o mecanismo de composições e travas.

Seguindo a cronologia do DHM, o Arakne [Bouvin 1999] é um modelo conceitual que está sendo desenvolvido também na universidade de Aarhus, Dinamarca, como resultado de uma tese de doutorado. O Arakne tem como objetivo modelar ferramentas para expansão da Web (web augmentation tools), ou seja, ferramentas que adicionam funcionalidades hipermídia à WWW. É importante observar que esse modelo irá incorporar os conceitos do mecanismo de notificação oferecidos pelo DHM na autoria de documentos na WEB.

5.4 CoVer e SEPIA

O CoVer [HAAKE 1992a] é um servidor hipermídia de versões implementado como uma extensão ao CHS (*Cooperative Hypermedia Server*) [STREITZ 1992] que, por sua vez, é baseado no HyperBase alemão [SCHUTT 1990]. O objetivo do CoVer, que forma a infra-estrutura de sistemas como o SEPIA [HAAKE 1993], é oferecer suporte à versões em ambientes de autoria hipermídia.

[34] Web Distributed Authoring and Versioning, trabalho que será comentado na Seção 5.5.

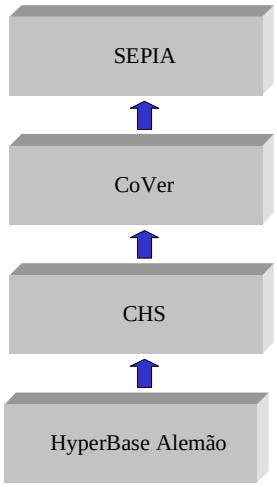


Figura 35: Relação entre SEPIA e sistemas antecessores

O CHS oferece persistência de nós, elos e composições, mas não preserva estados anteriores de objetos, deficiência contornada por funcionalidades de versionamento introduzidas no sistema CoVer, as quais foram implementadas no topo do CHS.

No CoVer, um objeto *O* é denominado de versionado quando *O* possui versões, e não versionado quando *O* não possui. O sistema representa objetos versionados pelos objetos denominados de estado múltiplo (*multi-state objects* – mobs), e objetos não versionados pelos objetos de estado único (*single-state objects* – snobs).

Um *mob* representa um objeto versionado armazenando todos os estados, chamados de versões e que podem ser nós, elos e composições do CHS, em seu conjunto de versões[35]. Um *snob* (nó, elo ou composição) pode ser criado no sistema e, eventualmente, tornado em um mob por uma operação explícita.

O CoVer oferece operações para derivação de versões de um determinado objeto. Além disso, cada versão de um mob pode ser congelada (*frozen*), de modo a preservar o estado dos objetos versionados. Seria interessante que essa mudança disparasse notificações a objetos relacionados, como versões derivadas e irmãs, porém,

[35] Esse conjunto é implementado como uma composição do CHS.

no entanto, isso não ocorre no CoVer, apesar do sistema congelar versões referenciadas implicitamente para preservação de estados de elos e composições.

O histórico de derivação de versões induz uma estrutura em grafo, independentemente da estruturação das versões em mobs. Para a conexão de versões ascendentes e descendentes, um elo específico, chamado de elo de derivação (*derivation link*), é utilizado. Versões de um mesmo mob conectadas por um elo de derivação são consideradas revisões de um objeto versionado.

Os autores do sistema argumentam que usuários editam um hiperdocumento para realizar uma determinada tarefa. Por exemplo, a edição de um hiperdocumento por um grupo de autores inclui tarefas como proposta de um sumário, criação de propostas alternativas, ou junção de diversas contribuições. Essas tarefas podem guiar uma criação automática de versões.

Enquanto mobs guardam o histórico de um único objeto versionado, as tarefas guardam os vários objetos (e suas versões) utilizados e criados para realização de um trabalho. Eventualmente, uma tarefa guarda o estado de um hiperdocumento que satisfaz os requisitos dos respectivos trabalhos. Qualquer objeto pode ser incluído em uma tarefa por uma operação de inclusão fornecida pelo CoVer. As alterações realizadas em objetos incluídos serão armazenadas automaticamente em versões derivadas: o CoVer congela os objetos incluídos, deriva novas versões desses objetos e efetua as alterações nessas versões derivadas.

Uma tarefa pode ser formada por várias subtarefas, e pode ser executada sequencialmente ou em paralelo. Assim, as tarefas de uma hierarquia de tarefas provêem um mecanismo para gerenciar a decomposição recursiva de processos de aplicação.

Dessa maneira, o CoVer estende a noção de objetos hipertextos para a noção de objetos hipertexto versionados, contudo, a aplicação tem de decidir como esses objetos devem ser versionados, isto é, as aplicações tem de implementar sua política específica de controle de versões. Consequentemente, o controle de notificação na política de versionamento de objetos também é de responsabilidade das aplicações.

O sistema SEPIA (*Structured Elicitation and Processing of Ideas for Authoring*) [HAAKE 1993] é um ambiente de autoria de documentos hipermídia que utiliza a infra-estrutura oferecida pelo CoVer.

Na verdade, as primeiras versões do sistema operavam diretamente sobre o CHS, não oferecendo funcionalidades para versionamento de objetos [STREITZ 1992] [HAAKE 1992]. Nessas primeiras versões, existia uma base de informações fornecida pelo CHS, que agrupava informações compartilhadas e particulares dos usuários. O SEPIA garantia uma visão comum dessa base de informações aos usuários, por meio do mecanismo de notificação de atualização do CHS. Trabalhos posteriores [HAAKE 1993] apontaram algumas deficiências desse sistema, como a falta de um histórico de um objeto, e introduziram a utilização do CoVer para a aquisição de facilidades para gerenciamento de versões no SEPIA. A versão do sistema que utiliza o CoVer é a que será comentada ao longo desta seção.

No SEPIA, quando vários autores desejam trabalhar simultaneamente em esboços separados de um mesmo documento, cada autor deve criar uma tarefa sucessora da tarefa que criou o documento original. O esquema de tarefas do CoVer preserva o documento original e mantém as alterações realizadas por cada autor em versões separadas que não interferem entre si. Usuários trabalhando em tarefas separadas são notificados sobre a criação de versões alternativas por outros usuários.

O sistema também oferece mecanismos para alteração simultânea de uma mesma versão *V* de um documento por um grupo de autores. Para controlar as alterações em objetos realizadas por diferentes autores em uma edição, o SEPIA armazena cada atualização em uma versão, desde que essa atualização não tenha sido efetuada pelo autor criador da versão, em uma nova versão derivada. Assim, revisões de objetos criadas por autores em colaboração são mantidas automaticamente no sistema. Nesse caso, observe que o estado original do objeto se perderia, caso o mesmo não fosse armazenado em uma cópia no sistema.

O SEPIA poderia utilizar conceitos como atributos versionáveis e não versionáveis do NCM para tentar evitar uma possível proliferação indesejável de versões (ou pelo menos guiar a derivação de versões) em atualizações concorrentes no

mesmo objeto. O mecanismo de notificação proposto para o NCM e seu controle de versões oferecem meios para reduzir esse problema, podendo o sistema de autoria indicar o comportamento desejado em atualizações concorrentes no mesmo objeto (armazenar ou não o histórico de atualizações). No NCM, cada usuário poderia ter uma única versão com atributos não versionáveis derivada do objeto a ser editado, e atualizações efetuadas pelos autores poderiam ser propagadas por meio do mecanismo de notificação. Caso houvesse a necessidade do histórico de atualizações das edições concorrentes, os atributos das versões poderiam ser tornados versionáveis.

Uma funcionalidade interessante do SEPIA é a possibilidade de investigação de versões de um objeto versionado: o usuário pode abrir um browser de mob e visualizar as relações de derivação de versões. A Figura 36 apresenta um exemplo de um browser de mob, onde versões de um mesmo objeto, que estavam incluídas nas tarefas "*Introduction*", "*Background*" e "CSCW", foram fundidas em uma versão, e essa incluída na tarefa "*Annotated Outline*".

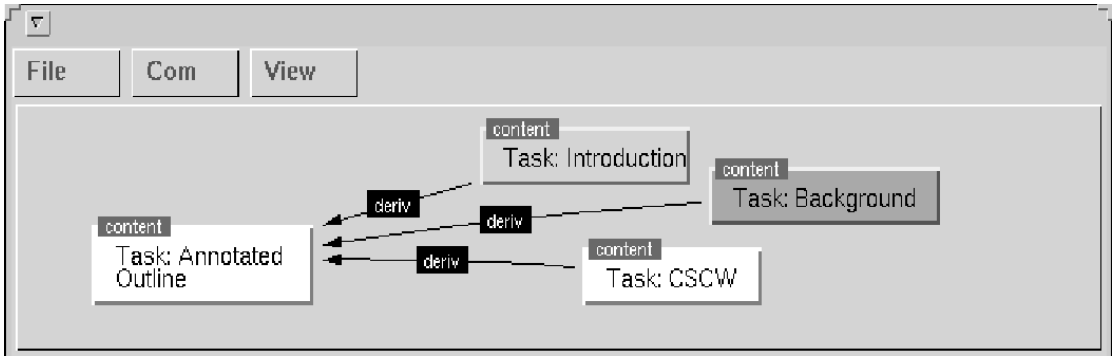


Figura 36: Visualização gráfica de versões por meio de um browser de mob no SEPIA

5.5 WebDAV

O grupo de trabalho do IETF (Internet Engineering Task Force), denominado WebDAV (Web Distributed Authoring and Versioning, também chamado de DAV), desenvolve extensões ao protocolo HTTP (Hypertext Transfer Protocol) para suportar autoria cooperativa de documentos na World Wide Web (WWW, Web) [WEBDAV 2000].

Hoje em dia, existem várias ferramentas heterogêneas com o mesmo objetivo, no entanto, por não seguirem um padrão, são incompatíveis entre si. A idéia principal do WebDAV é criar um protocolo padrão e funcional para a edição cooperativa de hiperdocumentos na Web.

Até agora, o grupo produziu uma única RFC [WEBDAV 1999] que descreve extensões ao HTTP/1.1. Essas extensões provêem um conjunto coerente de funções para:

1) Propriedades: A habilidade para criar, remover, e consultar informações sobre páginas Web, tais como autores, data de criação, etc.

2) Coleções: A habilidade de criar conjuntos de documentos relacionados. Um mesmo documento pode pertencer a várias coleções, seja diretamente ou por referência. As coleções podem ser pensadas como um meio de agrupamento de documentos, oferecendo funcionalidades simples como as de um diretório em um sistema de arquivos. Definições mais elaboradas como, por exemplo, a especificação de elos contextuais não é possível.

3) Travas (*Locks*): O controle de concorrência no WebDAV é obtido por meio de travas e possibilita que mais de um usuário trabalhe simultaneamente em um documento, sem que haja atualizações conflitantes.

4) Operações em Espaço de Nomes (*Namespace Operations*): A habilidade de instruir o servidor a copiar e mover recursos Web (arquivos html, gif, jpg, etc.).

Nessa RFC, é utilizada a XML[36] [BRAY 1998] para codificar parâmetros (entrada) e respostas (saída) de métodos, provendo vantagens como extensibilidade e internacionalização na entrada e saída de dados.

[36] XML (Extensible Markup Language) é uma meta-linguagem para especificação de linguagens para intercâmbio de documentos estruturados.

Existem algumas implementações de servidores e clientes, baseadas nesse RFC, disponíveis no momento. Uma implementação de servidor interessante é a que estende o servidor HTTP Apache [FIELDING 1997], por meio da adição de um módulo especial (mod_dav), para tratar e responder métodos WebDAV.

Facilidades para controle de versão e notificação ainda não estão completamente definidas, mas são pretensões importantes do grupo.

Em relação ao controle de versões, existe um "*draft*" criado pelo grupo que está em um estado avançado, próximo a se tornar uma versão estável. Na realidade, o trabalho de versões foi iniciado pelo grupo WebDAV e atualmente está sendo continuado por um subgrupo chamado Delta-V [DELTA-V 2000]. As facilidades principais do controle de versões no DAV incluem o registro do histórico de derivação de objetos, operações de *check-out* e *check-in*, atribuição de características de mutabilidade ou imutabilidade[37] à recursos, fusão de versões, recuperação de uma versão padrão (*default*) de um recurso e versionamento de coleções.

O controle de notificação ainda está em um estado inicial, somente foi produzido um "*draft*" com os seus requisitos, o qual expirou por volta de um ano e ainda não foi substituído. O mecanismo de notificação proposto para o NCM no Capítulo 3 poderia ser utilizado para contribuir na implementação do mesmo mecanismo no WebDAV. O controle de notificação apresentado, especialmente o esquema de inscrições automáticas no versionamento de objetos, é útil para modelos hipermídia baseado em composições e com controle de versões, que é o caso do DAV. Assim, usuários poderiam ser notificados quando novas versões de recursos HTTP fossem criadas, por exemplo. O modelo proposto facilitaria a definição de eventos para disparo de notificações, embora ainda fosse necessária a criação de um esquema escalável para transmissão e recepção de notificações disparadas segundo o paradigma *publish*/*subscribe*.

[37] Os atributos de um recurso imutável não podem ser modificados.

6. Considerações Finais

Esta dissertação apresentou o controle de versões e propôs um mecanismo de notificação para o NCM, ambos implementados pelo sistema de autoria hipermídia HyperProp. Com isso, o HyperProp tornou-se capaz de proporcionar meios para que documentos hipermídia sejam criados e editados simultaneamente por vários autores, onde o estado de edição dos documentos compartilhados pode ser divulgado para os autores pertinentes.

Várias dificuldades surgiram na implementação do controle de versões do NCM, que é bastante funcional, porém complexo, já que este é o primeiro trabalho a codificar o mecanismo de versionamento de acordo com as especificações mais recentes do NCM [SOARES 2000], incluindo o tratamento de versões de representação. Este é também o primeiro trabalho a codificar um mecanismo de notificação para o NCM.

Após a implementação do controle de versões e do esquema de notificação, algumas estratégias que utilizam os benefícios desses dois mecanismos foram sugeridas [Seção 4.4].

6.1 Implementação Anterior do NCM

A dissertação de mestrado [BATISTA 1994] implementou as classes do NCM especificadas na época, e grande parte do controle de versões do modelo, em C++ sobre o AIX versão 3.2 em estações IBM RISC System 6000. O trabalho apresentou uma proposta de notificação para o NCM, porém que não foi implementada, embora estruturas de dados e procedimentos para implementação tenham sido sugeridos na dissertação.

O sistema oferecia uma biblioteca de classes, determinadas pelo Modelo de Contextos Aninhados, que fornecia operações para criação de nós e elos (ambos não persistentes) em uma base privada.

O sistema implementado na época era monousuário. Diretrizes para tornar o sistema multiusuário foram, entretanto, dadas. Com isso, as operações de *check-out* e *check-in* foram substituídas por uma operação de criação de versões, que criava versões em uma base privada local e atualizava o contexto de versões correspondente. Era possível também ajustar atributos de contextos de versões, como o atributo "versão corrente" de um contexto de versões, por meio da operação de mudança de versão corrente, bem como operações para mudança de estado de um nó.

Não existia qualquer suporte gráfico para edição, como os mecanismos que hoje são oferecidos pelos *browsers* do sistema HyperProp [PINTO 2000].

O mecanismo de notificação sugerido por [BATISTA 1994] era operacional apenas para as bases privadas, agindo nos casos de atualização, mudança de estado e derivação de uma nova versão de um nó. Cada nó *N* possuía uma lista de referências invertida, que continha todos os nós que referenciavam[38] *N* e seriam notificados no caso de alteração de *N*, e uma lista de pertinência, que continha todos os contextos de usuário onde *N* estava contido e seriam notificados no caso em que *N* mudasse de estado ou derivasse uma nova versão.

Era difícil estender esse esquema de notificação para atender solicitações de nós a serem notificados pelo usuário, acréscimos de outros eventos que disparassem notificações e também novas notificações a serem enviadas em novas situações desejadas de versionamento.

Nessa proposta anterior, também não foi levado em consideração o estabelecimento de um agrupamento de nós para receberem notificações privativas.

6.2 Contribuições

Vários trabalhos correlatos existentes na literatura, alguns deles descritos no Capítulo 5, propõem um esquema de notificação de alteração simples: notificação em

[38] No trabalho de [Batista 1994], dizia-se que um nó *A* referenciava outro nó *B* quando *A* era origem de um elo cujo destino era *B*.

atualizações de conteúdos e em operações de manipulação de travas (*locks*). Apesar de alguns trabalhos oferecerem controle de versões, nenhum dos sistemas apresenta um esquema de notificação, em relação à operações de versionamento, como foi sugerido neste trabalho.

Alguns sistemas oferecem funcionalidades para inscrições em eventos de um determinado nó, mas não é possível estabelecer conexões semânticas em relação a características de derivação, como por exemplo, inscrição em eventos de atualização em filhos de derivação de um determinado nó. Além disso, não há mecanismos que possam fornecer um esquema automático de propagação (e poda) de notificações, baseados nessas mesmas conexões semânticas de derivação, como por exemplo, o repasse automático de uma mensagem de notificação a todos os descendentes, os quais repassam a seus descendentes, e assim sucessivamente.

A infra-estrutura oferecida pelo modelo de notificação proposto é poderosa, pois permite a adaptação desse mecanismo, baseado em inscrições avulsas e em regras de inscrição automáticas, para suprir diferentes necessidades. Além disso, essas próprias regras automáticas [Seção 3.3.5] sugerem o comportamento padrão de um sistema hipermídia para vários autores em uma edição cooperativa, o qual pode ser alterado conforme desejado. Esse comportamento pode também servir de ponto de partida para sistemas que queiram introduzir a funcionalidade de notificação em operações de versionamento, e também que necessitem especificar agrupamentos de nós que troquem notificações privativas.

Quanto ao controle de versões, a base de contexto de versões foi uma solução adequada para resolver o problema da representação da operação de fusão de versões, quando essas pertencem a contextos de versões diferentes, como comentado na Seção 2.3.3.2.

A implementação levantou várias questões importantes e detalhes a serem considerados, que, no modelo (versões e notificação), parecem irrelevantes, mas que, no entanto, revelam a distância que existe entre o modelo conceitual e sua implementação, conforme discutido no Capítulo 4.

Portanto, pode-se ressaltar as seguintes contribuições dadas por esta dissertação:

⇒ Concepção de um mecanismo de notificação extensível, adaptável e expressivo para modelos hipermídia baseado em composições, com controle de versões e que necessitem da troca de notificações privativas para agrupamento de nós;

⇒ Implementação do controle de notificação proposto, útil para a divulgação do estado de compartilhamento de documentos entre vários usuários.

⇒ Definição do nó base de contexto de versões no NCM;

⇒ Implementação do controle de versões no HyperProp, contemplando, dentre outras facilidades, a possibilidade de visualização gráfica do histórico de documentos;

6.3 Trabalhos Futuros

As regras automáticas de inscrição devem sofrer um refinamento sucessivo, para indicar quais delas realmente são as mais necessárias a um sistema de autoria hipermídia. Além disso, é interessante a adoção de uma linguagem para especificar essas regras automáticas, de modo que o modelo de notificação seja ainda mais expressivo e flexível.

Na implementação atual do controle de notificação, caso uma base privada, que está inscrita (por procuração) para receber notificações, não esteja disponível em um dado momento (indisponibilidade de rede, por exemplo), a notificação publicada irá se perder. Não existe um concentrador de mensagens de notificação para resolver esse caso. Talvez, uma outra versão do iBus, que realize a concentração de mensagens deva ser utilizada. Repare que esse caso é diferente da persistência da lista de inscrições [Capítulo 4], que está implementada e garante que um nó possa ser removido da memória principal (seja por acidente (*crash* do sistema) ou propositadamente) e da próxima vez que carregado em RAM, estar inscrito nos contextos de notificação pertinentes.

Um esquema de controle de acesso aos objetos NCM que leva em consideração a influência que nós de composição têm sobre seus nós internos, deve ser concebido. As estratégias de controle de acesso sugeridas por [FUKS 1994] podem ser aproveitadas para determinar os níveis de acesso à objetos no NCM, em conjunto com as influências dos nós de composição citadas acima. O controle de notificação pode ser estendido para capturar novos eventos de disparo, como a disponibilização da operação de derivação de versões de um determinado nó (o controle de acesso pode proibir que determinados nós derivem versões, por exemplo). Há também a possibilidade do controle de acesso permitir ou proibir inscrições de nós em determinados contextos de notificação, expandindo a funcionalidade inicial proposta pelo *container* de notificação para notificações privativas.

Mecanismos elaborados para consciência de grupo podem ser incluídos na versão atual do Sistema HyperProp. A consciência ou percepção de grupo (*group awareness*) envolve saber quem está utilizando o sistema e o que estão fazendo [DIX 1997], procedimento que pode ser auxiliado pelo próprio controle de notificação do sistema.

Futuramente, os atributos versionáveis e não versionáveis de nós terminais e de contexto de usuário no NCM poderão guiar o modo de edição de um determinado nó:

- Solicitação de edição concorrente em atributos versionáveis – Versões particulares são editadas em bases privadas e notificações de atualização são propagadas entre os usuários envolvidos à medida do necessário.

- Solicitação de edição concorrente em atributos não versionáveis – Controle de concorrência no acesso a um mesmo atributo de um nó, disparando notificações entre os usuários envolvidos.

Para o segundo caso, é preciso especificar mecanismos para o suporte à edição simultânea (usuários separados por uma rede de comunicação de dados) de um mesmo atributo não versionável de um determinado nó terminal ou de contexto de usuário.

Para isso, faz-se necessária a utilização de um mecanismo de controle de concorrência integrado ao mecanismo de notificação, que fornecerá subsídios para que o sistema de autoria hipermídia ofereça o ambiente adequado para execução de tal tarefa. Esse controle de concorrência pode ser, por exemplo, baseado em travas, onde operações de obtenção e liberação de travas podem ser comunicadas por meio do controle de notificação, que, nesse caso, terá de contemplar adicionalmente esses dois eventos de disparo.

Apêndice A: Ciclos em Grafo de Versões

A inserção manual de um elo *E* em um contexto de versões ou em uma base de contexto de versões, deve ser submetida a um algoritmo para detectar se *E* irá tornar o grafo de versões (grafo direcionado) cíclico.

O algoritmo utilizado na implementação para decidir se uma determinada inserção é válida ou não para o sistema, é conhecido na literatura [CHRISTOFIDES 1975] e é descrito abaixo.

A idéia principal é que se um grafo é acíclico, então ele deve possuir pelo menos um vértice com grau de saída igual a zero (sem arestas de saída), denominado de vértice *folha*. Essa condição é necessária, porém não é suficiente. Se uma folha *F* em um grafo acíclico for removida (e também as arestas que chegam em *F*), então o grafo resultante também será acíclico. Caso continue-se removendo folhas, um dos dois casos acontecerá:

i) Eventualmente todos os vértices serão removidos, significando que o grafo é acíclico.

ii) O grafo não estará vazio e não existirão mais folhas. Nesse caso, o grafo é cíclico.

Então, para testar se um grafo é cíclico, executa-se o seguinte procedimento que sempre termina:

1) Se o grafo não tem vértices, pare. O grafo é **acíclico**.

2) Se o grafo não tem folhas, pare. O grafo é **cíclico**.

3) Escolha uma folha *F* no grafo. Remova *F* e todas as arestas que chegam em *F*, resultando em um novo grafo.

4) Vá para o passo 1.

Apêndice B: Contextos de Notificação

Dado um nó *N*, os possíveis contextos de notificação que *N* pode publicar são:

(*N*, *Derivação de Versão*),(*N*, *Check-in*), (*N*, *Permanência*), (*N*, *Obsolescência*), (*N*, *Inserção de um Nó*), (*N*, *Remoção de um Nó*), (*N*, *Atualização de Atributos Não Versionáveis*), (*N*, *Atualização de Atributos Versionáveis*), (*N*, *Inserção de Elo*), (*N*, *Remoção de Elo*), (*N*, *Adição de Novos Atributos*), (*N*, *Destruição)*;

(*N*, *Recebimento de Notificação* (*Ancestral*, *Derivação de Versão*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Check-in*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Permanência*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Obsolescência*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Inserção de um Nó*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Remoção de um Nó*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Atualização de Atributos Não Versionáveis*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Atualização de Atributos Versionáveis*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Inserção de Elo*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Remoção de Elo*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Adição de Novos Atributos*)), (*N*, *Recebimento de Notificação* (*Ancestral*, *Destruição*));

(*N*, *Recebimento de Notificação* (*Ascendente*, *Derivação de Versão*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Check-in*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Permanência*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Obsolescência*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Inserção de um Nó*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Remoção de um Nó*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Atualização de Atributos Não Versionáveis*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Atualização de Atributos Versionáveis*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Inserção de Elo*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Remoção de Elo*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Adição de Novos Atributos*)), (*N*, *Recebimento de Notificação* (*Ascendente*, *Destruição*));

(*N*, *Recebimento de Notificação* (*Descendente*, *Derivação de Versão*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Check-in*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Permanência*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Obsolescência*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Inserção de um Nó*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Remoção de um Nó*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Atualização de Atributos Não Versionáveis*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Atualização de Atributos Versionáveis*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Inserção de Elo*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Remoção de Elo*)), (*N*, *Recebimento de Notificação* (*Descendente*, *Adição de Novos Atributos*)), (*N*, *Recebimento de Notificação*(*Descendente*, *Destruição*));

(*N*, *Recebimento de Notificação* (*Independente*, *Derivação de Versão*)), (*N*, *Recebimento de Notificação* (*Independente*, *Check-in*)), (*N*, *Recebimento de Notificação* (*Independente*, *Permanência*)), (*N*, *Recebimento de Notificação* (*Independente*, *Obsolescência*)), (*N*, *Recebimento de Notificação* (*Independente*, *Inserção de um Nó*)), (*N*, *Recebimento de Notificação* (*Independente*, *Remoção de um Nó*)), (*N*, *Recebimento de Notificação* (*Independente*, *Atualização de Atributos Não Versionáveis*)), (*N*, *Recebimento de Notificação* (*Independente*, *Atualização de Atributos Versionáveis*)), (*N*, *Recebimento de Notificação* (*Independente*, *Inserção de Elo*)), (*N*, *Recebimento de Notificação* (*Independente*, *Remoção de Elo*)), (*N*, *Recebimento de Notificação* (*Independente*, *Adição de Novos Atributos*)), (*N*, *Recebimento de Notificação* (*Independente*, *Destruição*));

(*N*, *Recebimento de Notificação* (*Ancestral*, *Recebimento de Notificação*))

Bibliografia

[BAPAT 1996] Bapat, A.; Wäsch, J.; Aberer, K.; Haake, J. “HyperStorM: An Extensible Object-Oriented Hypermedia Engine”. Proceedings of the Seventh ACM Conference on Hypertext, 1996.

[BATISTA 1994] Batista, T. V. “Controle de Versões no Modelo Hipermídia de Contextos Aninhados”. Dissertação de Mestrado do Departamento de Informática da PUC-Rio, Agosto 1993.

[BENTLEY 1997] Bentley, R. et al. “Basic Support for Cooperative Work on the World Wide Web”. International Journal of Human-Computer Studies, 46(6), Junho 1997.

[BLIP 1998] “Basic Lightweight Information Protocol”. http://www.blip.org, Dezembro 1998.

[BOOCH 1996] Booch, G.; Jacobson, I.; Rumbaugh, J. “The Unified Modeling Language for object-oriented Development”, *Documentation Set Version 0.91 Addendum UML Update*, 1996.

[BOUVIN 1999] Bouvin, N. O. “Unifying Strategies for Web Augmentation”. Proceedings of the ACM Conference on Hypertext, 1999.

[BRAY 1998] Bray, T.; Paoli, C. M.; Sperberg-McQueen. “Extensible Markup Language (XML)”. World Wide Web Consortium Recommendation REC-xml-19980210.

[CARVALHO 1999] Carvalho, C. A. A. “Um Ambiente para Apoio ao Trabalho em Grupo na WWW”. Dissertação de Mestrado do Departamento de Informática da PUC-Rio, Setembro, 1999.

[CASANOVA 1991] Casanova, M. A.; Tucherman, L.; Lima, M. J.; Rangel Netto, J. L.; Rodriguez, N. R.; Soares, L. F. G. “The Nested Context Model for Hyperdocuments”. Proceedings of Hypertext 91, Texas, 1991.

[CERI 1997] Ceri, S.; Fraternali, P. “Designing Database Applications with Objects and Rules – The IDEA Methodology”. Addison Wesley, 1997.

[CHOU 1986] Chou, H. “A Unifying Framework for Version Control in a CAD Environment”. Proceedings of the Twlfth International Conference on Very Large Databases, 1986.

[CHRISTOFIDES 1975] Christofides, N. "Graph Theory: An Algorithmic Approach".

Academic Press, 1975.

[CORBA 1998] "CORBA Notification Service". Object Management Group, Inc. (OMG), 1998.

[CROWLEY 1990] Crowley, T.; Milazzo, P. “MMConf: An Infrastructure for Bulding Shared Multimedia Applications”. Proceedings of the Conference on Computer Supported Cooperative Work, Outubro, 1990.

[CVS 1998] “Concurrent Versions System”. http://www.cyclic.com/cyclic-pages/overview.html, Agosto 1998.

[DEBRA 2000] De Bra, P. M. E. “Hypermedia Structures and Systems”. On-Line Course on World Wide Web, Eindhoven University of Technology. http://wwwis.win.tue.nl/2L690, Março 2000.

[DELTA-V 2000] "IETF Delta-V Working Group: Extending the Web with versioning and configuration management". http://www.webdav.org/deltav, Julho 2000.

[DIX 1997] DIX, A. “Challenges for Cooperative Work on the Web: An Analytical Approach”. Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW ’97), 1997.

[ELLIS 1991] Ellis, C. A.; Gibbs, S. J.; Rein, G. L. “Groupware: some issues and experiences”. Communications of the ACM 34, 1991.

[ENGELBART 1973] Engelbart, D. C.; Watson, R. W.; Norton, J. C. “The Augmented Knowledge Workshop”. Proceedings of the National Computer Conference, 1973.

[ENGELBART 1984] Engelbart, D. C. “Collaboration Support Provisions in Augment”. Proceedings of AFIPS ’84, Los Angeles, CA, Fevereiro, 1984.

[ENGELBART 1990] Engelbart, D. “Knowledge-Domain Interoperability and an Open Hyperdocument System”. Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW ‘90), Outubro, 1990.

[FERRET 1997] Ferret, R. “Access 97 Essentials : Academic Version”, John Preston, 1997.

[FIELDING 1997] Fielding, R.; Kaiser, G. “The Apache HTTP Server Project”, IEEE Internet Computing, 1(4), 1997.

[FUKS 1994] Fuks, H.; Moura, L. M. “A Document Based Approach for Cooperation”. Journal of Brazilian Computer Society,

Number 1, Vol 1, July 1994.

[GRALA 1997] Grala, A.; Heuser, C. “GDOC: A System for Storage and Authoring of Documents through WEB Browsers”. Proceedings of the International Conference of the Chilean Computer Science Society, 1997.

[GREENBERG 1999] Greenberg, S. Roseman, M. “Groupware Toolkits for Synchronous Work”. Computer-Supported Cooperative Work, Trends in Software Series 7, John Wiley & Sons, 1999.

[GRONBAEK 1994] Gronbaek, K.; Hem, J. A.; Madsen, O. L.; Sloth, L. “Cooperative Hypermedia Systems: a Dexter-based architecture. Communications of the ACM, 37(2), Fevereiro, 1994.

[GRONBAEK 1997] Gronbaek, K. “Designing Dexter-based Hypermedia Services for the World Wide Web”. Proceedings of the ACM Conference on Hypertext, 1997.

[GRONBAEK 1999] Gronbaek, K. ; Trigg, R. H. "From Web to Workplace: Designing Open Hypermedia Systems". MIT Press, 1999.

[HAAKE 1992] Haake, J.; Wilson, B. “Supporting Collaborative Writing of Hyperdocuments in SEPIA”. Proceedings of the ACM Conference on Computer-Supported Cooperative Work (CSCW ‘92), 1992.

[HAAKE 1992a] Haake, A. “CoVer: A Contextual Version Server for Hypertext Applications”. Proceedings of the 4th ACM conference on Hypertext, 1992.

[HAAKE 1993] Haake, A. ; Haake J. M. “Take CoVer: Exploiting Version Support in Cooperative Systems.” Proceedings of the InterCHI '93, Amsterdam, Netherlands, April 26-29, 1993.

[HAAKE 1997] Haake, J. M.; Wang, W. “Flexible Support for Business Processes: Extending Cooperative Hypermedia with Process Support. Proceedings of the International Conference on Supporting on Supporting Group Work: the Integration Challenge, 1997.

[HAAKE 1998] Haake, J.; Wang, W. “Collaboration Support in Open Hypermedia Environments”. Proceedings of the Fourth Workshop on Open Hypermedia Systems (HT’ 98). ACM, 1998.

[HALASZ 1988] Halasz, F.; “Reflections on Notecards: Seven Issues for the Next Generation of Hypermedia Systems”. Communications of the ACM, 31, 7 (Julho), 1988.

[HALASZ 1994] Halasz, F.; Schwartz, M. “The Dexter Hypertext Reference Model (Edited)”. Communications of the ACM, 37(2), Fevereiro, 1994.

[HATHAWAY 1999] Hathaway, B. “Comp.Object.FAQ Version 1.0.9 (04-02) Part 6/13”. http://www.faqs.org/faqs/object-faq/part6, Setembro 1999.

[HETHMON 1997] Hethmon, P. S. “Illustrated Guide to HTTP”. Manning Publications, 1997.

[IBEX 1995] IBEX Object Systems SA. “ITASCA Distributed Object Database Management System”. Technical Summary, 1995.

[IBUS 1998] “iBus: Programmer’s Manual”. http://www.softwired-inc.com, Abril 1998.

[JAVA 2000] “The Java Programming Language”. http://java.sun.com, Julho 2000.

[JMS 1998] “Java Message Service”. http://java.sun.com/products/jms, Abril 1998.

[JSDT 1998] “Java Shared Data Toolkit User Guide”. http://www.sun.com/software/jsdt, Abril 1998.

[MARSHALL 1991] Marshall, C. C.; Halasz, F. G.; Rogers, R. A.; Janssen Jr, W. C. “Aquanet: a Hypertext Tool to Hold Your Knowledge in Place”. Proceedings of Hypertext ’91, Dezembro, 1991.

[MATHUR 1995] Mathur, A. G.; Hall, R. W.; Jahanian, F.; Prakash, A.; Rasmussen, C. “The Publish/Subscribe Paradigm for Scalable Group Collaboration Systems”. Department of Electrical Engineering and Computer Science, University of Michigan, Technical Report, 1995.

[MHEG 1995] MHEG. “Informatio Technology – Coded Representation of Multimedia and Hypermedia Information Objects – Part1: Base Notation”. *Committee Draft ISO/IEC CD 13522-1*. Julho 1995.

[MINOR 1993] Minor, S.; Magnusson, B. “A Model for Semi-(a)Synchronous Collaborative Editing”. Proceedings of the Third European Conference on Computer Supported Cooperative Work, Kluwer Academic Publishers, 1993.

[MUCHALUAT 2000] Muchaluat, D. C.; Soares, L. F. G. "Linguagens de Descrição de Arquitetura x Modelos Hipermídia". Relatório Técnico do Laboratório TeleMídia, Departamento de Informática da PUC-Rio, 2000.

[NORONHA 1998] Noronha, M. A.; Golendziner, L. G.; Santos, C. S. “Compartilhamento de Componentes com Versões em Documentos Estruturados”. Anais do Simpósio Brasileiro de Banco de Dados, 1998.

[OHS 1999] “Open Hypermedia Systems Working Group”. http://www.ohswg.org, Outubro 1999.

[PENG 1993] Peng, C. “Survey of Collaborative Drawing Support Tools: Design Perspectives and Prototypes”. Computer-Supported Cooperative Work, 1(3), 1993.

[PIMENTEL 1998] Pimentel, M. G.; Kutova, M. A. S.; Macedo, A. A.; Teixeira, C. A. C.; “Hiperdocumentos Estruturados no Suporte ao Trabalho Cooperativo em Sistemas Abertos Distribuídos”. Anais do Semish, 1998.

[PINTO 2000] Pinto, L. A. "Autoria Gráfica de Estruturas de Documentos Hipermídia no Sistema HyperProp". Dissertação de Mestrado a ser defendida no Departamento de Informática PUC-Rio, Agosto de 2000.

[POET 1998] POET SDK Java Edition. “POET 5.1 Programmer’s Guide”, 1998.

[RAPOSO 1999] Raposo, A. B.; Magalhães, L. P.; Ricarte, I. L. M. “Interação na WEB”. Anais da JAI ’99, 1999.

[REINERT 1998] Reinert, Ó. H.; “Cooperative Authoring using Open Spatial Hypermedia”. Master Thesis, Aarhus University, Dinamarca, 1998.

[RODRIGUES 1998] Rodrigues, R. F. Muchaluat-Saade, D. C, Soares, L. F. G. "Composite Nodes, Contextual Links and Graphical Structures Views on the WWW". Special Issue on World Wide Web of the Journal of Brazilian Computer Society, Vol. 5, N 2, Nov 1998

[ROSEMAN 1992] Roseman, M.; Greenberg, S. “GROUPKIT: A Groupware Toolkit for Building Real-Time Conferencing Applications”. Proceedings of the Conference on Computer Supported Cooperative Work, Outubro-Novembro, 1992.

[SCHIMDT 1992] Schimdt, K; Bannon, L. J.; “Taking CSCW Seriously – Supporting Articulation Work”. Proceedings of the Conference on Computer Supported Cooperative Work, October-Novembro, 1992.

[SCHUCKMANN 1996] Schuckmann, C.; Kirchner, L. et al. “Designing Object-Oriented Synchronous Groupware with COAST”.

Proceedings of the Conference on Computer Supported Cooperative Work, Novembro, 1996.

[SCHUTT 1990] Schutt, H.; Streitz, N. A. “HyperBase: A Hypermedia Engine Based on a Relational Database Management System”. Proceedings of the European Conference on Hypertext, 1990.

[SHIRMOHAMMADI 1998] Shirmohammadi, S.; de Oliveira, J.C.; Georganas, N.D. “Applet-Based Telecollaboration: A Network Centric-Approach”. IEEE Multimedia, 5(2), Abril/Junho, 1998.

[SMIL 1998] W3C Recommendation. "Synchronized Multimedia Integration Language (SMIL) 1.0 Specification". http://www.w3.org/TR/REC-smil, Junho 1998

[SMITH 1996] Smith, J. D. “EColabor: Collaborative Elaboration of Documents”. NTT Multimedia Communications Laboratories, 1996.

[SOARES 1995] Soares, L. F. G.; Rodriguez, N. L.; Casanova, M. A. “Nested Composite Nodes and Version Control in an Open Hypermedia System”. Information Systems, Special Issue on Multimedia Information Systems, v. 20, n. 6, 1995.

[SOARES 1999] Soares, L.F.G.; Souza, G. L.; Rodrigues, R.; Michaluat, D. “Versioning Support in the HyperProp System”. Multimedia Tool & Application, Vol. 8, No. 8, 1999.

[SOARES 2000] Soares, L. F. G. “Modelo de Contextos Aninhandos”, Relatório Técnico do Laboratório TeleMídia, Departamento de Informática da PUC-Rio, 2000.

[SPRINGER 1990] Springer, G. "Scheme and the Art of Programming". MIT Press and McGraw Hill, 1990.

[STEFIK 1987] Stefik, M. et al. “WYSIWIS revised: Early Experiences with Multiuser Interfaces”. ACM Transaction on Office Information Systems, 5(2), Abril, 1987.

[STREITZ 1992] Streitz, N. A.; Haake, J. M.; Hannemann, J.; Lemke, A.; Schuler, W.; Schütt, H. A.; Thüring, M.”SEPIA: A Cooperative Hypermedia Authoring Environment”. Proceedings of the 4th ACM Conference on Hypertext (ECHT'92), 1992.

[STREITZ 1994] Streitz, N. A.; Geibler, J.; Haake, J. M.; Hol, J. “DOLPHIN: Integrated Meeting Support across Local and Remote Desktop Environments and Liveboards. Proceedings of the Conference on Computer Supported Cooperative Work, Outubro, 1994.

[STREITZ 1994a] Streitz, N. A. “Putting Objects to Work: Hypermedia as the Subject matter and the Medium for Computer-Supported Cooperative Work”. ECOOP ’94: Object-Oriented Programming, Lecture Notes in Computer Science, Julho, 1994.

[SYBASE 1999] “The Sybase Relational Database Management System”. http://www.cs.indiana.edu/database/Sybase/Sybase.html, Novembro 1999

[SZWARCFITER 1984] Szwarcfiter, J. "Grafos e Algoritmos Computacionais". Editora Campus, 1984.

[TAKAHASHI 1996] Takahashi, K. Higuchi, M. “Hypermedia Support for Workflow Management in Collaborative Document Production”. NTT Software Laboratories, 1996.

[TESCH 1995] Tesch, T.; Wäsch, J. “Transaction Support for Cooperative Hypermedia Document Authoring – A Study on Requirements”. Proceedings of the 8th EDRG Workshop on Database Issues and Infrastructure in Cooperative Information Systems (EDRG-8), Trondheim, Norway, Agosto, 1995.

[TICHY 1985] Tichy, W. “RCS – A System for Version Control”. Software Practice and Experience, Vol. 15, Junho, 1985.

[WANG 1998] Wang, W.; Haake, J. M. “Flexible Coordination with Cooperative Hypermedia. Ninth ACM Conference on Hypertext and Hypermedia: Links, Objects, Time and Space Structure in Hypermedia Systems, Junho, 1998.

[WEBDAV 1999] RFC 2518. "HTTP Extensions for Distributed Authoring -- WebDAV", 1999.

[WEBDAV 2000] “The Web Distributed Authoring and Versioning Working Group”. http://www.webdav.org, Julho 2000.

[WHITEHEAD 1999] Whitehead, E. J. Jr.; Goland, Y. Y. “WebDAV: A Network Protocol for Remote Collaborative Authoring on the Web”, 1999.

[WIDOM 1996] Widom, J.; Ceri, S. “Active Database Systems: Triggers and Rules for Advanced Database Processing”. Morgan Kaufmann Publishers, 1996.

[WIIL 1992] Wiil, U. K.; Legget, J. J. “HyperForm: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems”. ACM Conference on Hypertext, 1992.

[WIIL 1992a] Wiil, U. K; Legget, J. J. “Hyperform: An Extensible

Hyperbase Managemente System". Technical Report TAMU-HRL 92-003, Texas A&M University, 1992.

[WIIL 1993] Wiil, U. K. "Experiences with HyperBase: A Hypertext Database Supporting Collaborative Work". SIGMOD, 22(4), Dezembro, 1993.

[WIIL 1993a] Wiil, U.K.; Legget, J. J. "Concurrency Control in Collaborative Hypertext Systems". Proceedings of the ACM Conference on Hypertext, 1993.

[WIIL 1993b] Wiil, U. K. "Issues in the Design of EHTS: A Multiuser Hypertext System for Collaboration". Technical Report, 1993.

[WIIL 1997] Wiil, U.K.; Legget, J. J. "Workspaces: the HyperDisco Approach to Internet Distribution". Proceedings of the Eighth ACM Conference on Hypertext, 1997.

[WIIL 1997a] Wiil, U. K; Legget, J. J. "Hyperform: A Hypermedia System Development Environment". ACM Transactions on Information Systems, 15, 1, janeiro de 1997.

[WIIL 1998] Wiil, U. K.; Nürnberg, P. J. "Collaboration in Open Hypermedia Environments". Technical Report, 1998.

[WILSON 1994] Wilson, P. "Introducing CSCW – What is It and Why We Need It". Computer-Supported Cooperative Work – The Multimedia and Networking Paradigm, Unicom Seminars Ltd., 1994.

[WINSTON 1988] Winston, P. H.; Horn, B. K. P. "Lisp". Addison-Wesley Pub Co, 1988.