Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Forby: Disseminação de notificações sobre ficheiros partilhados

Pedro Miguel do Rosário Ferreira de Sousa Nº 26205

Orientador: Prof. Doutor Nuno Manuel Ribeiro Preguiça

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Engenharia Informática

Lisboa
20 de Fevereiro de 2009

Agradecimentos

Quero agradecer a todos aqueles que, directa ou indirectamente, contribuíram para a elaboração deste trabalho. Este trabalho foi parcialmente suportado pelo Projecto Files EveryWhere (Project #POSC/EIA/59064/2004) e por uma bolsa de investigação do Centro de Informática e Tecnologias da Informação da FCT-UNL.

Em particular, gostaria de agradecer ao meu orientador, Nuno Preguiça, pela confiança, disponibilidade, conselhos, dedicação e constante apoio demonstrado ao longo deste trabalho. O impacto que teve neste trabalho é enorme.

Ao pessoal do 254, em especial ao Pedro Amaral, Nuno Luís, João Soares e David Navalho, agradeço a sua disponibilidade, excelentes conversas e trocas de ideias, que me ajudaram a enriquecer este trabalho.

Ao Filipe Grangeiro, pela amizade, conversas e apoio dado, não só ao longo da elaboração deste trabalho, mas também durante todo o meu percurso académico. Este apoio foi muito importante para concluir esta etapa da minha vida.

A todos os meus outros amigos e amigas, que não vou nomear por ser uma lista relativamente longa, pela amizade, paciência e compreensão.

À Leonor, pela paciência, incentivo e compreensão demonstrados, ao longo dos últimos anos e, em particular, durante este trabalho. O seu apoio foi fundamental para me manter focado nos meus objectivos.

Em especial, quero agradecer aos meus pais e ao meu irmão Sérgio, todo o carinho, compreensão e incentivo que me deram, não só durante este trabalho, mas ao longo da minha vida. Este apoio incondicional tornou possível chegar até este ponto e por isso lhes estou muito agradecido.

Resumo

O avanço e disponibilidade das tecnologias de comunicação, nomeadamente a Internet, tornaram possível a utilização deste meio como forma de colaboração entre diversos indivíduos. Assim, é cada vez mais comum encontrar grupos de pessoas que utilizam os mesmos ficheiros ou que pretendem partilhar conjuntos de ficheiros.

Esta situação leva à necessidade de se desenvolverem sistemas que suportem estes grupos de utilizadores. Este suporte pode passar por várias alternativas, entre as quais manter diferentes réplicas dos ficheiros existentes, nas diversas máquinas, automaticamente actualizadas ou apenas notificar os utilizadores sobre os acessos a esses ficheiros.

Neste trabalho apresenta-se o Forby, um sistema genérico de recolha e disseminação de informação relativa ao estado dos ficheiros partilhados. Este sistema, de fácil integração e utilização, permite o desenvolvimento de aplicações que lidam com estes ficheiros de forma simples, uma vez que fornece os mecanismos para monitorização e partilha de informação.

O Forby fornece um mecanismo eficiente e extensível para fazer a monitorização da área de trabalho dos utilizadores, recolhendo eventos associados às chamadas ao sistema de ficheiros.

Para a disseminação de eventos, o grupo de utilizadores é organizado numa rede *peer-to-peer* que permite a troca de informação por todos os participantes. Dado que se tratam de ficheiros partilhados, é possível optimizar o tráfego que passa pela rede, minimizando o número de mensagens trocadas entre os participantes, sem comprometer o bom funcionamento da aplicação.

Para fazer a avaliação do Forby, implementaram-se duas aplicações distintas que tiram partido das funcionalidades oferecidas. A primeira, o P2PVC^2 estende os sistemas de controlo de versões CVS/SVN com informação de *awareness* possibilitando que cada participante do grupo tenha consciência dos ficheiros que estão a ser modificados por outros utilizadores.

A segunda aplicação permite replicar conjuntos de ficheiros presentes numa directoria pelos vários elementos do grupo, podendo ser utilizada, por exemplo, para que vários utilizadores partilhem as suas fotografias de forma simples e automática.

Palavras-chave: Disseminação de Eventos; Ficheiros Partilhados; Sistema de Awareness; Informação Contextual;

Abstract

The improvement and availability of communication technologies, including the Internet, has allowed wide area collaboration between several individuals. Therefore, it is common to find groups of people who share files.

This leads to the need to implement systems to support these groups of participants. Different types of support can be implemented, such as automatically keeping different file replicas up to date, in several computers, or simply notifying users about the state of files in other replicas.

This dissertation presents Forby, a generic system for monitoring and disseminating information about shared files. The system is easy to use and integrate with any application that manages shared files, providing mechanisms to monitor the file system activities and share this information among users.

Forby offers an extensible and efficient way to monitor the user's working area, collecting events associated with the file system calls executed in that area.

Concerning the event dissemination, the workgroup participants are organized in a peer-to-peer network that allows the exchange of information among them. Since the events reflect the access to the shared files, it is possible to optimize the flow of information propagated through the network, thus minimizing the number of exchanged messages.

To evaluate Forby, two distinct applications were developed. The first, P2PVC^2 extends version control systems with awareness information that allows users to keep aware of the activity of other users.

The second application replicates sets of files, which are kept in a given directory in each computer. This application can be used, for instance, by users that want to share photos in an easy and automatic way.

Keywords: Event Dissemination; Shared Files; Awareness System; Contextual Information;

Conteúdo

Lista de Figuras **13**

Lista de Tabelas **15**

1 Introdução **17**

1.1 Introdução geral e descrição do problema 17

1.2 Solução apresentada 19

1.3 Principais contribuições 20

1.4 Organização da dissertação 21

2 Trabalho relacionado **23**

2.1 Sistemas de controlo de versões 23

2.1.1 RCS 23

2.1.2 CVS 24

2.1.3 SVN 25

2.1.4 Google Docs 26

2.1.5 Resumo 27

2.2 Sistemas de *awareness* 27

2.2.1 VC^2 28

2.2.2 State Treemap 30

2.2.3 Palantír 31

2.2.4 Integração do CVS com notificações e chat 32

2.2.5 Resumo 33

2.3 Monitorização do sistema de ficheiros 34

2.3.1 Windows Driver Kit: API Installable File Systems Drivers 34

2.3.2 inotify 34

2.3.3 JNotify 35

2.3.4 Resumo e Comparação 36

2.4 Sistemas de disseminação de eventos 36

2.4.1 Hermes 37

2.4.2 Elvin com suporte para ambientes móveis 39

3 Desenho do sistema **43**
3.1 Objectivos e Requisitos 43
3.2 Arquitectura 45
3.2.1 Sistema de detecção 46
3.2.2 Sistema de disseminação 47

4 Implementação **49**
4.1 Ambiente 49
4.2 Sistema de detecção 49
4.2.1 Event Receiver 49
4.2.1.1 Aplicações de monitorização externas 51
4.2.2 Event Filter 52
4.2.3 Event Manager 53
4.3 Sistema de disseminação 54
4.3.1 Organização dos participantes 54
4.3.2 Sistema de ponto de encontro 55
4.3.2.1 Servidor Central 56
4.3.2.2 Rede sobreposta estruturada (Pastry) 57
4.3.3 Redes públicas e privadas 58
4.3.4 Gestão de falhas 58
4.3.5 Dissemination Events e as suas propriedades 59
4.3.5.1 Repositórios remotos 61
4.3.6 *Caching*, sincronização e suporte para desconexão 62
4.3.7 Outros aspectos importantes 64

5 Avaliação **65**
5.1 Avaliação Qualitativa 65
5.1.1 Aplicações 65
5.1.1.1 P2PVC^2 65
5.1.1.2 Aplicação de replicação 68
5.2 Avaliação Quantitativa 70
5.2.1 Sistema de detecção 70
5.2.2 Sistema de disseminação 73

6 Conclusões e trabalho futuro **79**
6.1 Conclusões 79
6.2 Trabalho Futuro 80

Bibliografia **86**

Lista de Figuras

2.1 Conflito com necessidade de resolução manual. 24
2.2 Notificação apresentada pelo VC^2 ao abrir um ficheiro em utilização 29
2.3 Árvore Original 31
2.4 Visualização em Treemap 31
2.5 Janela onde são apresentados os eventos gerados por novas versões 33
2.6 Organização dos filtros segundo a sua altura 35
2.7 Ligação entre os dois tipos de componentes do sistema Hermes 38

3.1 Componentes do Forby. 45
3.2 Organização dos módulos do sistema de detecção. 46

4.1 Interface de um FSEvent. 50
4.2 Módulo de detecção implementado com JNotify. 51
4.3 Módulo de detecção implementado com *socket* UDP. 51
4.4 Interface do EventFilter 52
4.5 Interacção entre Event Manager e aplicação. 54
4.6 Servidor central como ponto de encontro. 56
4.7 DHT como ponto de encontro. 57
4.8 Ligação entre nós públicos e privados. 59
4.9 Alterações feitas ao repositório. 60

5.1 Métodos de *commute* e *overwrite* definidos para o P2PVC2. 67
5.2 Inicialização do sistema de detecção. 67
5.3 Métodos de *commute* e *overwrite* para eventos to tipo FILE_WRITE 69
5.4 Métodos de *commute* e *overwrite* para eventos to tipo FILE_DELETE 69

Lista de Tabelas

2.1 Resumo dos sistemas de controlo de versões. 27
2.2 Resumo dos sistemas de *awareness*. 33

5.1 Tempos de execução e *overhead* para os diferentes tipos de monitorização. 71
5.2 Tempos de execução médio para uma escrita. 72
5.3 Tempos de execução médio para uma leitura. 72
5.4 Número de comunicações periódicas. 75
5.5 Número de comunicações numa operação de escrita. 76
5.6 Número de comunicações numa operação de escrita. 76

1. Introdução

O avanço e disponibilidade das tecnologias de comunicação, nomeadamente a Internet, tornaram possível a utilização deste meio como forma de colaboração entre diversos indivíduos. Assim, é cada vez mais comum encontrar grupos de pessoas que utilizam os mesmos ficheiros ou que pretendem partilhar conjuntos de ficheiros.

Existem inúmeras aplicações que necessitam de gerir ficheiros partilhados. Um exemplo deste tipo de aplicações pode ser um sistema de gestão de fotografias partilhadas, onde os utilizadores recebem informação sobre as fotografias geridas. Com essa informação, os participantes podem optar receber automaticamente as novas fotografias adicionadas ao sistema ou serem informados da sua existência.

Outro exemplo é o dos sistemas de *awareness*, onde se procura oferecer um suporte para o trabalho colaborativo, para que todos os participantes do grupo de trabalho tenham uma visão global do estado do sistema e saibam quais os ficheiros que estão a ser modificados pelos outros participantes, em qualquer instante.

Além das aplicações específicas, para suportar a partilha de ficheiros, os utilizadores usam outras aproximações, como por exemplo, recorrendo a mecanismos de comunicação explícitos como o correio electrónico ou *instant messaging*, soluções de ficheiros partilhados, como o Google Docs [14], ferramentas de sincronização (CVS [4] e SVN [28]) ou sistemas de gestão de dados distribuídos ([21, 30]).

Todos estes exemplos permitem às aplicações gerir os ficheiros partilhados. No entanto, estas soluções não se adequam a todas as aplicações que utilizam estes ficheiros, dado que pretendem resolver problemas específicos. Outro aspecto a considerar é a necessidade de intervenção do utilizador para partilhar a informação, situação que ocorre em alguns dos exemplos anteriores.

1.1 Introdução geral e descrição do problema

A utilização de ficheiros partilhados levanta diversos problemas relativos à forma como é feita a recolha, manutenção e disseminação da informação relativa ao acesso e modificação destes ficheiros, por parte de cada utilizador.

Dado que se tratam de ficheiros partilhados, uma grande parte das aplicações que utilizam

estes ficheiros têm necessidade de manter um estado global coerente, em todos os nós do sistema. Caso contrário, a visão de cada um dos utilizadores poderia ser desactualizada e levar à introdução de erros, no que diz respeito à informação que é mantida nos respectivos ficheiros.

Existem diversas classes de aplicações que operam sobre ficheiros partilhados, como por exemplo, ferramentas de replicação de informação por vários nós de um grupo, sistemas de controlo de versões ou aplicações de *awareness* que procuram melhorar a experiência do trabalho cooperativo.

Algumas aplicações, como os sistemas de controlo versões CVS ou SVN permitem sincronizar os ficheiros a pedido. Esta aproximação pode levar a problemas devido ao esquecimento de actualizar os ficheiros e à falta de informação sobre a actividade dos outros participantes.

Outras aplicações, como o Palantír [25], têm necessidade de fornecer, de forma contínua e em todo o sistema, a informação sobre o acesso e modificação de cada um dos ficheiros da área de trabalho.

Outro exemplo deste tipo de aplicações é o VC^2 [17], um sistema de *awareness* que pretende evitar conflitos derivados de alterações concorrentes dos mesmos ficheiros, antes de estes serem propagados para um servidor de controlo de versões. Para implementar estas aplicações é necessário recorrer a mecanismos que permitam monitorizar os ficheiros partilhados e disseminar a informação recolhida sobre os acessos e modificações desses ficheiros de forma simples e eficiente.

A ideia geral é associar um evento a cada acção do utilizador que seja significativa, no sentido de poder provocar uma alteração ao estado sistema. Como exemplo, normalmente atribui-se um evento a uma criação ou alteração de um ficheiro, já que essa informação pode ser importante para os outros utilizadores.

O sistema deve depois permitir a propagação destes eventos para os nós do sistema interessados neles. A acção a efectuar aquando da recepção dos eventos dependerá depois da aplicação, podendo resultar, por exemplo, na notificação do utilizador.

Outro factor que está normalmente associado à utilização de ficheiros partilhados é a componente assíncrona do seu uso, ou seja, é possível ter utilizadores a aceder a ficheiros em horários diferentes, o que dificulta a gestão feita pelas aplicações. Torna-se então importante garantir que a informação gerada durante os períodos de desconexão possa ser acedida em qualquer altura, por qualquer utilizador.

Estes factores são determinantes no desenho de qualquer aplicação que lide com ficheiros

partilhados, o que leva a que, no geral, o processo de implementação seja bastante complexo.

1.2 Solução apresentada

O trabalho realizado nesta dissertação procurou criar um suporte para lidar, de forma simples, com os problemas apresentados anteriormente. Para isso, foi criado o Forby, um sistema genérico que permite recolher e disseminar, de forma eficiente, eventos gerados por acessos ao sistema de ficheiros e que pode ser utilizado por qualquer aplicação que necessite destas funcionalidades.

O Forby foi desenvolvido a pensar nas aplicações que necessitam de gerir ficheiros partilhados por um grupo de utilizadores, tentando simplificar a sua implementação. Para isso, é fornecida uma API simples de utilizar, que gere toda a informação necessária à monitorização e disseminação de eventos, evitando que os programadores tenham que se preocupar com estes dois aspectos.

No que diz respeito à monitorização, pretendeu-se fornecer um mecanismo eficiente, genérico e extensível para recolher os eventos do sistema de ficheiros. A utilização em vários sistemas de operação foi uma preocupação, pelo que a solução apresentada pode facilmente ser integrada em qualquer ambiente, com mínimo impacto para a aplicação final.

Relativamente à disseminação, o objectivo é organizar os participantes de um grupo para que seja possível partilhar a informação recolhida em cada nó, de forma eficiente. O sistema de disseminação procura minimizar as mensagens enviadas pela rede tirando partido das propriedades inerentes aos ficheiros partilhados Adicionalmente, procura trocar mensagens apenas entre os nós interessados em recebê-las.

O sistema oferece um mecanismo simples de suporte à desconexão, permitindo que um participante tenha acesso à informação disseminada no período em que não se encontrava ligado ao sistema.

Outro aspecto importante do sistema de disseminação desenvolvido é a sua adaptação aos recursos existentes. Por exemplo, a existência de um servidor remoto pode ajudar na organização da rede e no armazenamento dos dados, mas no caso de este não existir, o sistema mantém-se funcional.

Adicionalmente, o sistema permite suportar nós presentes em diferentes redes privadas, sem necessidade de alterar as configurações da rede.

Ao ser genérico, o sistema Forby pode ser utilizado em qualquer tipo de aplicação. A utilização do sistema é simples, facilitando o desenvolvimento das aplicações, uma vez que encapsula toda a informação relativa à monitorização e disseminação, passando para a aplicação apenas os dados necessários ao seu funcionamento.

Além dos exemplos mencionados anteriormente, este sistema poderia ser usado em muitas outras aplicações. Por exemplo, poderia servir de suporte a um sistema de notificações, em que o fim de uma tarefa fosse verificado pela monitorização de ficheiros específicos (como em [15]). Após este facto, o sistema de disseminação poderia ser usado para iniciar a próxima tarefa.

1.3 Principais contribuições

A principal contribuição deste trabalho é a criação de um sistema genérico de monitorização e disseminação de eventos optimizado para ficheiros partilhados, procurando tirar partido da semântica dos eventos recolhidos para tornar as aplicações mais eficientes.

A utilização do Forby permite o desenvolvimento de aplicações de forma mais simples, uma vez que toda a complexidade relativa à recolha e troca de informação é tratada pelo sistema.

No âmbito do trabalho, podem considerar-se outras contribuições mais específicas. Primeiro, o desenvolvimento de mecanismos de monitorização da área de trabalho dos utilizadores para diferentes sistemas de operação e a sua comparação com outros sistemas já existentes, no que diz respeito ao *overhead* imposto.

Segundo, a criação de um mecanismo de disseminação de notificações flexível e configurável, com suporte para desconexão e que tira partido da informação recolhida sobre os ficheiros partilhados para optimizar as mensagens enviadas pela rede de utilizadores.

Por fim, as aplicações implementadas utilizando o Forby - o P2PVC^2 e uma aplicação de replicação - oferecem ao utilizador ferramentas simples e eficientes para lidar com os problemas que pretendem resolver. No caso do P2PVC^2, criou-se uma aplicação escalável de disseminação de informação de *awareness* e no caso da aplicação de replicação, fornece-se um mecanismo simples para replicar ficheiros em várias máquinas.

1.4 Organização da dissertação

Após esta introdução sobre o contexto do trabalho efectuado e a solução apresentada, no capítulo seguinte apresenta-se um levantamento do estado da arte relacionada com este trabalho. No capítulo 3, encontra-se o desenho do sistema, onde se referem os objectivos e funcionalidades oferecidas. No capítulo 4, descreve-se a implementação do Forby, seguindo-se a avaliação do mesmo, no capítulo seguinte. Por fim, no capítulo 6, indicam-se as conclusões finais e as perspectivas de trabalho futuro.

2. Trabalho relacionado

Neste capítulo apresenta-se o estado da arte relacionado com o trabalho desenvolvido. Em primeiro lugar são abordados alguns sistemas de controlo de versões e reconciliação. Na secção 2.2 faz-se uma apresentação de alguns sistemas de *awareness*. De seguida, apresentam-se alguns mecanismos de monitorização do sistema de ficheiros e por fim referem-se sistemas de disseminação de eventos.

2.1 Sistemas de controlo de versões

Os sistemas de controlo de versões permitem que seja feita uma gestão das diferentes versões de qualquer documento digital, sendo normalmente utilizados por equipas de desenvolvimento de projectos, dado que é normal a necessidade de pessoas diferentes modificarem os mesmos ficheiros de forma concorrente.

Para um bom funcionamento deste tipo de sistemas é essencial ter um mecanismo que consiga fazer a reconciliação das modificações, assim como manter um histórico com todas as alterações feitas ao longo do tempo para que facilmente se aceda a uma versão antiga, em caso de necessidade.

2.1.1 RCS

O Revision Control System [29] foi um dos primeiros sistemas de controlo de versões a ser implementado tendo como objectivo automatizar o processo de armazenamento, recolha, modificação e reconciliação de qualquer tipo de ficheiros.

O RCS opera sobre ficheiros individuais e organiza as versões numa árvore, criando uma nova folha para cada versão nova. Para armazenar de forma eficiente todas as versões existentes, cada nível da árvore guarda apenas as diferenças para a versão anterior de forma transparente para o utilizador.

A organização em árvore permite que se alterem os ficheiros concorrentemente, ficando assim cada ramo com uma versão diferente. Existe também a possibilidade de definir *locks* que impedem o desenvolvimento em simultâneo da mesma versão.

Para se utilizar o sistema é necessário, em primeiro lugar, criar um repositório (fazendo

check-in de um ficheiro). De seguida é possível fazer o *check-out* do ficheiro, efectuar as alterações necessárias e finalmente fazer *check in* de novo.

O processo de actualização de versões, seja ele no repositório ou no sistema de ficheiros local, pode originar conflitos. Um conflito ocorre quando existem modificações concorrentes num dado ficheiro, ou seja, quando dois ou mais utilizadores alteram o mesmo ficheiro concorrentemente e pretendem reconciliar as versões.

Para a resolução de conflitos é necessário ter em conta as alterações efectuadas. No caso de terem sido modificadas linhas diferentes do mesmo ficheiro, a unificação dos ficheiros é feita automaticamente, resultando numa versão com todas as alterações. No caso de serem modificadas as mesmas linhas, é pedido ao utilizador para resolver manualmente o problema, sendo disponibilizadas as duas versões.

A Figura 2.1 apresenta um exemplo de um conflito que deve ser resolvido manualmente já que ambos os utilizadores mudaram a mesma linha.

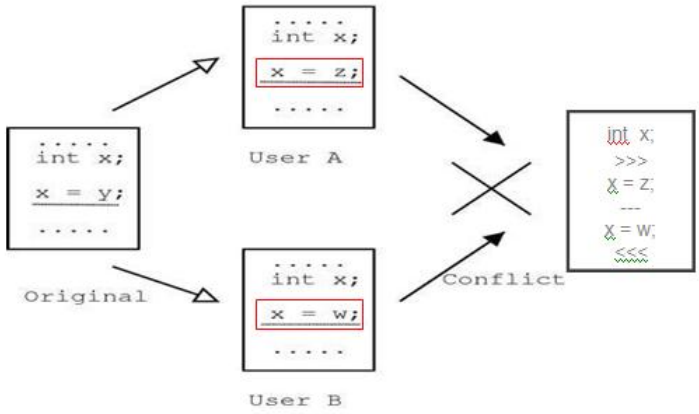

Figura 2.1 Conflito com necessidade de resolução manual.

Este sistema tem algumas limitações como o facto de operar sobre ficheiros isolados, não sendo possível trabalhar com um projecto inteiro (grupos de ficheiros) e apenas funciona localmente a uma máquina.

2.1.2 CVS

O Concurrent Versioning System [4] é um sistema que estende o RCS, conseguindo resolver algumas das limitações mais significativas deste sistema, como a possibilidade de gerir não

só ficheiros isolados mas também projectos e permitindo que estes sejam acedidos de forma remota.

O CVS funciona num modelo cliente/servidor, sendo o servidor utilizado como repositório para os ficheiros e o seu histórico. Como no RCS, para utilizar o sistema é necessário que os clientes acedam ao repositório e façam *check-out* dos ficheiros.

O processo de *check-out* cria uma cópia local, de todos os ficheiros presentes no projecto, no sistema de ficheiros do utilizador, possibilitando que cada utilizador altere de forma concorrente os mesmos ficheiros sem afectar o repositório central. Quando as alterações são concluídas, é necessário fazer o *check-in* dos ficheiros locais para o servidor para que os restantes utilizadores tenham acesso à nova versão.

Para garantir que não se perdem alterações em versões concorrentes o servidor apenas permite que se façam modificações sobre a última versão presente no repositório. Para isso, antes de se confirmar o *commit*, o cliente verifica se as alterações propostas estão feitas sobre a última versão presente no servidor. No caso de outro utilizador ter actualizado a versão do ficheiro, é feita a actualização da versão local, reconciliando a nova versão disponível no servidor com as modificações efectuadas localmente.

Os conflitos existentes devem ser resolvidos como explicado no RCS. Feita esta verificação é então possível fazer o *commit* da nova versão que contém as alterações feitas localmente pelo utilizador.

Quando uma versão é adicionada ao servidor incrementa-se um contador de versão associado ao ficheiro alterado e associa-se um campo ao histórico com a identificação do autor e a data da modificação. Também é possível adicionar comentários a cada versão, que podem ser consultados por qualquer utilizador.

O CVS tem também integrado um mecanismo de partilha de informação de *awareness*, o CVS *Watch*. Ao definir um *watch* um utilizador pode receber informação sobre a utilização de um determinado ficheiro que esteja interessado, sempre que alguém efectua alguma operação sobre esse ficheiro. Para que estas notificações sejam enviadas, o cliente tem que, ao modificar um ficheiro, invocar explicitamente um comando para indicar esse facto.

2.1.3 SVN

O Subversion [28] é outro sistema de controlo de versões que foi criado com o objectivo de se tornar numa alternativa mais robusta ao CVS.

Os pontos fortes deste sistema, em relação a outros sistemas de controlo de versões (em particular o CVS), são uma maior velocidade de acesso aos repositórios, a criação de versões quando os ficheiros são apagados ou movidos, *commits* verdadeiramente atómicos, que garantem a consistência das versões dos repositórios e a manutenção do histórico de revisões para os ficheiros que são renomeados, copiados, movidos ou apagados [33].

Para garantir *commits* atómicos, neste sistema é utilizado o conceito de transacção. Uma transacção define um conjunto de operações sobre um grupo de ficheiros. Cada transacção começa sobre uma versão existente no sistema, tendo associado um ramo específico na árvore de versões. As transacções ou são aceites e se tornam na versão mais recente ou são abortadas.

O uso de transacções faz com que seja possível actualizar, de forma atómica, grupos de ficheiros em vez de estes serem tratados individualmente. O utilizador pode escolher os ficheiros que fazem parte da transacção e tem a garantia que todos eles são transferidos para o servidor ou para o sistema local. Caso a transacção fique a meio, as alterações são descartadas.

O SVN utiliza a mesma arquitectura (cliente/servidor) e o mesmo modo de funcionamento (*check-out* para o sistema de ficheiros local, alterações, *commit* com controlo de conflitos) que foi explicado no CVS.

2.1.4 Google Docs

O Google Docs [14] permite controlar modificações concorrentes efectuadas por vários utilizadores. O sistema tem um servidor central onde se cria um documento (texto, folha de calculo ou apresentação) para ser partilhado.

Os clientes são convidados pelo autor do documento a ligar-se ao servidor e a fazerem as alterações *online*. O servidor encarrega-se de disseminar as notificações pelos clientes em tempo real em vez de cada participante alterar localmente uma versão e no final fazer o *commit*.

Tomando como exemplo uma folha de cálculo, cada um dos participantes terá um cursor de cor distinta e pode mover-se pelas células do documento. Enquanto se efectua uma alteração, essa célula fica bloqueada para não existirem conflitos e assim que se confirma a modificação, esta é imediatamente vista pelos outros utilizadores.

No caso de se tratar de uma folha de texto, a informação de alteração simultânea é dada por um aviso e caso se modifiquem as mesmas linhas concorrentemente, o último utilizador a propagar a modificação é informado do conflito e deve resolvê-lo como entender.

2.1.5 Resumo

Na Tabela 2.1 apresenta-se um resumo dos pontos que podem ser considerados importantes para uma comparação entre os sistemas de controlo de versões apresentados.

Dos sistemas apresentados, os três primeiros são os mais relevantes para este trabalho, dado que gerem ficheiros partilhados dos quais os utilizadores mantêm uma réplica local. Apesar de estes sistemas usarem uma arquitectura baseada num servidor central, existem sistemas com propriedades semelhantes, baseados em arquitecturas distribuídas (e.g. Bazaar [3], SVK [27]). Existem ainda outros sistemas que permitem a sincronização par-a-par de ficheiros, como o Unison [22].

O que é interessante observar nestes sistemas é a necessidade do utilizador iniciar explicitamente a propagação da informação, o que pode levar a erros e esquecimentos.

O sistema desenvolvido nesta tese permite, por exemplo, monitorar as modificações efectuadas localmente e poderia ser usado para questionar o utilizador sobre a sua vontade de propagar as notificações efectuadas.

Sistema	Arquitectura	Modo de Interacção	Actualização da informação
RCS	Local	Assíncrona	Manual
CVS	Cliente/Servidor	Assíncrona	Manual
SVN	Cliente/Servidor	Assíncrona	Manual
Google Docs	Cliente/Servidor	Síncrona	Automática

Tabela 2.1 Resumo dos sistemas de controlo de versões.

2.2 Sistemas de *awareness*

No suporte ao trabalho cooperativo, o fornecimento de informação de *awareness* permite uma melhor coordenação entre as actividades dos vários colaboradores [8]. Essa informação pode ser fornecida por sistemas de *awareness* autónomos ou integrados em outras ferramentas.

Neste contexto, um aspecto importante para os grupos de utilizadores que pretendem coordenar o trabalho a desenvolver é que cada um dos participantes tenha, a qualquer momento, a

informação das tarefas que os restantes utilizadores já realizaram ou estão a desempenhar nesse momento.

Para tal, alguns destes sistemas integram mecanismos que permitem a um colaborador verificar o estado dos objectos do sistema, quando foram utilizados, se existem modificações concorrentes, entre outros tipos de informação contextual.

Alguns sistemas (e.g. [12]) também integram mecanismos de suporte à comunicação entre participantes, que é essencial no desenvolvimento de qualquer projecto em grupo.

Outra possibilidade para permitir a comunicação entre os participantes de um projecto passa pela utilização de sistemas gerais de colaboração que integram diversas funcionalidades de troca de informação como *chats*, fóruns ou repositórios de dados.

Alternativamente, podem ser utilizadas aplicações de comunicação *single-user* genéricas, como o correio electrónico, para comunicar.

Outra alternativa passa por levar a colaboração até ao contexto da aplicação, ou seja, embutir nas aplicações funcionalidades extra que permitam a troca de experiências. Esta aproximação permite que os utilizadores continuem a usar as suas aplicações preferidas. A esta ideia dá-se geralmente o nome de colaboração contextual [12].

A colaboração contextual tem várias vantagens sobre outros métodos de colaboração. A principal vantagem é, possivelmente, permitir que os utilizadores se mantenham concentrados no seu trabalho e não percam tempo com troca de contexto entre aplicações ou a aprender a utilizar outras ferramentas.

Um segundo benefício é a possibilidade de juntar, a cada parte do trabalho, registos sobre as considerações de cada utilizador, como por exemplo as ideias que surgiram ou aquilo que falta fazer em vez de as propagar por *email* ou *chat*, situação que por vezes leva a que se perca o contexto dado que não existe uma ligação explícita entre as notas e o texto que referem.

Seguem-se alguns exemplos de sistemas de *awareness* que procuram melhorar o desempenho de equipas que necessitam de se coordenar durante a realização de um determinado projecto.

2.2.1 VC^2

O sistema VC^2 [17] baseia-se num modelo de cliente/servidor e executa sobre um servidor de CVS ou SVN, estendendo a utilização destes dois sistemas de controlo de versões com a possibilidade de disseminar informação de *awareness*.

O objectivo desta ferramenta é notificar os membros de um grupo de trabalho se existem ou não alterações a serem efectuadas a cada momento, mesmo antes de ser feito o commit das alterações finais, reduzindo-se os conflitos originados pelo trabalho concorrente. Deste modo, os utilizadores mantêm uma visão global do estado do sistema, minimizando a necessidade de sincronizar manualmente os ficheiros com o repositório e resolver os seus conflitos.

Para apresentar esta informação aos utilizadores, o VC^2 utiliza uma ferramenta de monitorização da área de trabalho do sistema de controlo de versões e, sempre que há uma modificação num ficheiro dessa área, adiciona um registo num ficheiro de meta-informação que posteriormente é guardado no servidor. O servidor guarda a informação de quem está a modificar os ficheiros, possibilitando que essa informação seja consultada por qualquer utilizador.

Quando um utilizador acede a um ficheiro, o ficheiro de meta-informação presente no servidor é consultado, verificando-se se a versão é a mais actual e se existem outros utilizadores a usar esse ficheiro, o que pode originar um conflito. Nesse caso, é possível pedir ao utilizador que está a modificar o ficheiro para submeter as suas alterações, caso assim o pretenda (Figura 2.2).

Figura 2.2 Notificação apresentada pelo VC^2 ao abrir um ficheiro em utilização (figura retirada de [17].

Apesar de serem atingidos os objectivos principais, ou seja, permitir fornecer informação de *awareness* com um impacto mínimo para o utilizador usando um sistema controlo de versões já implementado e sem necessidade de fazer alterações ao servidor, este sistema não é escalável.

O principal problema é o servidor utilizado não ser activo, o que faz com que os clientes tenham necessidade de estar constantemente a fazer pedidos ao servidor, em vez de apenas receberem a informação de *awareness* necessária. Uma vez que não se pretende fazer alterações

no servidor utilizado, é necessário encontrar outra forma de difundir esta informação, para minimizar o *polling* feito ao servidor.

Neste contexto, a utilização do Forby pode melhorar a performance do VC^2, no que diz respeito à sua escalabilidade, evitando a necessidade de utilizar o servidor para disseminar a informação de *awareness*.

2.2.2 State Treemap

O State Treemap [19] é um elemento gráfico (*widget*) desenhado com ajuda de um grupo de arquitectos para permitir que os utilizadores tenham conhecimento das acções realizadas sobre os documentos partilhados pelo grupo. Este *widget* foca-se no problema de coordenação entre os diversos membros da equipa de desenvolvimento e mostra graficamente a divergência de estados dos objectos distribuídos. O State Treemap está integrado numa plataforma que suporta grupos virtuais de arquitectos.

Este sistema foi desenvolvido para um ambiente assíncrono, permitindo que existam múltiplos fluxos de modificações em simultâneo, que as modificações feitas num fluxo só sejam propagadas para os outros quando o utilizador as validar e que uma alteração visível não implica que os outros fluxos sejam imediatamente alterados.

Para visualizar o estado do sistema são usados Treemaps que facilitam a visualização de árvores com elevado número de objectos. Cada folha corresponde a um rectângulo e as áreas dos rectângulos são definidas de acordo com o tamanho do objecto que mapeiam. Nas Figuras 2.3 e 2.4, apresenta-se um exemplo da forma como os elementos são mapeados. A Figura 2.3 apresenta uma árvore que pode corresponder à organização de um sistema de ficheiros.

Cada folha representa um ficheiro enquanto os nós representam as directorias. Tendo em conta que cada ficheiro tem um tamanho associado, é possível mapear essa informação num rectângulo (Figura 2.4), conseguindo-se assim inferir sobre o tamanho de cada objecto sem ter que percorrer toda a árvore. Neste exemplo, o ficheiro E1 é mais pequeno que o F2 e o C3 é o maior ficheiro presente no sistema.

O princípio do State Treemap é preencher os rectângulos da árvore de acordo com o estado dos objectos. Os estados possíveis são: actual, modificado localmente, modificado remotamente, necessita de actualização, potencial conflito e existe conflito.

Todos estes estados são exclusivos, ou seja, não é possível um objecto ter dois estados

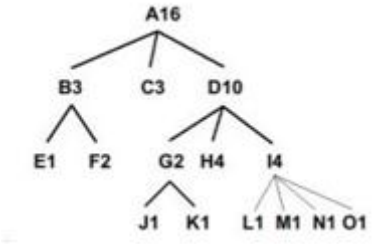

Figura 2.3 Árvore Original(figura retirada de [19].

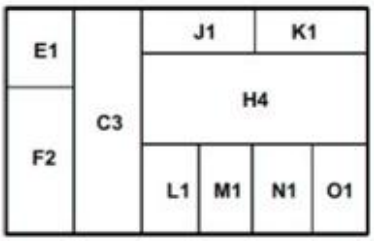

Figura 2.4 Visualização em Treemap(figura retirada de [19].

distintos em simultâneo. Cada um deles é apresentado no Treemap com uma cor distinta permitindo ao utilizador diferenciar os estados de forma intuitiva.

Os Treemaps permitem que o utilizador veja o sistema no seu todo, o que o leva a ter uma maior sensibilidade para a evolução do mesmo, sem ter que efectuar qualquer operação sobre o sistema.

2.2.3 Palantír

O sistema de *awareness* Palantír [25] permite que os seus utilizadores tenham informação do que se passa nas áreas de trabalho de outros elementos do seu grupo. Em particular, informa-os sobre quem modifica os objectos, calcula aproximadamente quais as potenciais consequências das alterações e apresenta toda essa informação graficamente tentando não interferir com o trabalho do utilizador.

Para permitir que todos os participantes sejam notificados das alterações que vão sendo efectuadas, o sistema Palantír usa uma aproximação que o diferencia dos outros sistemas, que corresponde à contínua monitorização do espaço de trabalho, em vez de disseminar notificações apenas quando é feito um *check in* ou *check out*. Para isso, a informação dos objectos sobre

os quais se está a trabalhar é extraída nas várias acções efectuadas no sistema de controlo de versões e é disseminada frequentemente.

Ao utilizar esta aproximação de disseminar permanentemente a informação, o cliente não tem necessidade de se sincronizar manualmente com os outros clientes recorrendo ao repositório uma vez que recebe as notificações dos outros clientes quando estas são geradas.

No Palantír, a área de trabalho encontra-se dividida em várias subáreas que podem ter cada uma a sua configuração. Para cada uma delas existe um monitor que se encarrega de enviar notificações aos restantes utilizadores dessas subáreas. Quando recebidos, estes eventos são interpretados e posteriormente apresentados ao utilizador para que este tenha uma informação contínua sobre o estado do sistema.

O Palantír pode ser integrado numa aplicação que permita trabalhar em grupo (por exemplo o CVS) e centra-se na distribuição, organização e apresentação da informação considerada importante nessas aplicações.

2.2.4 Integração do CVS com notificações e chat

Em [11] apresenta-se uma extensão ao CVS para que este passe a integrar a capacidade de difundir notificações sobre modificações em tempo real e de permitir a comunicação entre os membros do grupo através de *chat*.

Com esta extensão, os autores procuraram mostrar que é possível introduzir, num sistema de apoio ao desenvolvimento em grupo, funcionalidades que permitem substituir com vantagem diversas ferramentas normalmente utilizadas pelos programadores como o correio electrónico ou as salas de *chat*, através de um simples mecanismo de notificações e de uma interface multi-funções ao nível do utilizador.

Numa primeira fase deste trabalho, os autores estudaram a utilização do CVS em conjunto com uma *mailling-list* onde iam disseminando os comentários sobre as alterações que cada elemento fazia no código.

Segundo os participantes, apesar de ser interessante, esta aproximação requeria um maior esforço por parte dos leitores para perceber as alterações e como estas se reflectem na versão que foi submetida no repositório.

Para melhorar a eficiência da colaboração em grupo, evitando que seja necessário recorrer ao *email* para difundir e receber a informação das modificações nos ficheiros, decidiu-se utilizar o sistema Elvin [9] para difundir essa informação. O sistema Elvin é um sistema genérico de

disseminação de eventos que permite que sejam enviadas notificações para todos os elementos de um grupo.

A ideia base do sistema implementado é ter uma janela semelhante à da Figura 2.5 em cada cliente e sempre que há uma alteração de versão no servidor, é gerado um evento com o *log* associado à nova versão. Este evento é difundido por todos os participantes e é apresentado na janela para que estes saibam em tempo real que houve uma alteração dos ficheiros do projecto.

Figura 2.5 Janela onde são apresentados os eventos gerados por novas versões (figura retirada de [11].

Com esta simples extensão consegue-se melhorar a colaboração entre membros de um grupo que utilizem o CVS como gestor de ficheiros do projecto em desenvolvimento.

2.2.5 Resumo

Além dos sistemas apresentados, cujas propriedades mais importantes são resumidas na Tabela 2.2, têm sido propostas muitas outras soluções. Por exemplo, em [13] os autores integram a informação de *awareness* num editor de texto mantendo algum nível de privacidade. Em [6], a informação de *awareness* é apresentada no contexto de um editor de texto, através de notificações.

O sistema desenvolvido neste trabalho poderia ser utilizado na implementação de qualquer um destes sistemas, simplificando o seu processo de desenvolvimento. Na maioria dos sistemas utilizar-se-iam as duas funcionalidades de monitorização e disseminação.

Sistema	Extracção da Informação	Apresentação da Informação (como)	Apresentação da Informação (quando)
VC[2]	Ao aceder aos ficheiros	Aplicação própria	Ao receber um evento sobre um ficheiro em uso
State Treemap	Ao aceder aos ficheiros	Visualização de *treemaps*	Ao consultar os *treemaps*
Palantír	Ao aceder aos ficheiros (operação CVS)	Aplicação própria	Sempre que é gerado um evento
CVS com notificações e chat	Utilizando o *log* do CVS	Aplicação própria (*tickertape*)	Sempre que é gerado um evento

Tabela 2.2 Resumo dos sistemas de *awareness*.

2.3 Monitorização do sistema de ficheiros

Algumas ferramentas de monitorização de sistemas de ficheiros permitem, entre outras funcionalidades, recolher informação sobre o acesso ou modificação dos ficheiros presentes numa determinada área. No geral, estes sistemas permitem que uma aplicação (ou utilizador) seja notificada quando é efectuada uma operação sobre um ficheiro monitorizado.

Nesta secção apresentam-se algumas formas de fazer a monitorização de sistema de ficheiros para recolha de eventos interessantes para uma aplicação.

2.3.1 Windows Driver Kit: API Installable File Systems Drivers

O sistema Windows tem um módulo - Installable File Systems Driver API [18] - que permite a integração de filtros no sistema operativo, os quais intersectam as chamadas ao sistema de ficheiros. Estes filtros têm o nome de *minifilters*. A API fornecida para a criação de filtros simplifica o desenvolvimento de *drivers*, dado que muitas das operações complexas relativas à interacção com o sistema de ficheiros são oferecidas. Desta forma evita-se a implementação de um *driver* de raiz, processo que se considera bastante complicado.

O *driver* de controlo dos filtros, o Filter Manager é activado quando um *minifilter* é carregado. O filtro regista-se no *driver*, indicando o tipo de operações de I/O que pretende receber. Quando uma chamada ao sistema de ficheiros interessante é gerada, a informação é enviada para o filtro, para ser processada.

A passagem da informação por cada filtro activo é feita de forma sequencial, sendo a ordem determinada por um parâmetro designado por altitude. Por exemplo, se existirem dois filtros, um de um anti-vírus e outro de replicação, a altitude do primeiro deve ser superior, para que a informação seja analisada pelo anti-vírus antes de ser replicada.

A Figura 2.6 apresenta os passos seguidos por uma chamada ao sistema de ficheiros com três filtros carregados.

2.3.2 inotify

O inotify [32] é um sistema de notificações sobre alterações de ficheiros desenvolvido para Linux. Este sistema permite que se faça a monitorização de ficheiros individuais ou directorias. Adicionado ao *kernel* desde a sua versão 2.6.13, veio substituir o anterior mecanismo de

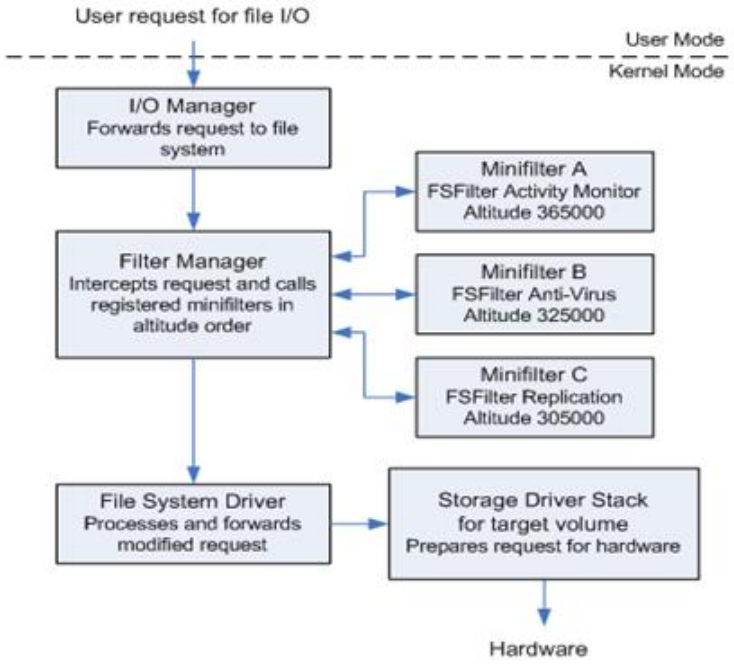

Figura 2.6 Organização dos filtros segundo a sua altura (figura retirada de [18]).

monitorização designado por **dnotify**.

Para uma aplicação utilizar este sistema, deve definir os ficheiros que pretende monitorizar e, através da análise dos *inodes* associados aos ficheiros, são recolhidas notificações sobre as alterações efectuadas, que são enviadas para a aplicação. O inotify recolhe informação sobre diversos tipos de eventos, como *open*, *close*, *read*, *write*, *create*, *delete*, etc.

O inotify mantém, para cada *inode* referente a um ficheiro monitorizado, uma lista dos processos interessados em receber informação sobre as alterações/acessos a esse *inode*. Quando existe uma interacção com o ficheiro, é enviada uma notificação para os processos interessados.

2.3.3 JNotify

O JNotify [1] é uma biblioteca que permite a uma aplicação Java recolher eventos do sistema de ficheiros, que funciona tanto em ambiente Linux como em Windows. A monitorização feita em Linux utiliza a API do Inotify para a recolha de eventos, encapsulando as notificações geradas por este sistema. Para fazer a recolha de notificações em Windows, é utilizada a API win32.

Os eventos definidos no JNotify são: file *create*, *modified*, *renamed* e *delete*. Estes são os eventos comuns a ambos os ambientes. No entanto é possível recolher eventos específicos de cada sistema operativo, como file *open* no Linux, à custa de portabilidade do código da aplicação.

Para usar esta biblioteca, a aplicação deve definir os eventos que pretende recolher para a área a monitorizar. Depois, deve registar-se como receptora das notificações e sempre que algum evento é gerado, a aplicação é notificada.

2.3.4 Resumo e Comparação

Os sistemas apresentados nas secções anteriores permitem monitorizar e capturar eventos relativos ao acesso ao sistema de ficheiros. O inotify faz a captura dos eventos gerados no sistema Linux. A biblioteca JNotify utiliza a API do inotify para a captura dos eventos em Linux e a API Win32 em Windows. Os *minifilters* recolhem informação sobre as chamadas ao sistema em Windows.

Comparando o JNotify/inotify com os *minifilters*, pode afirmar-se que os filtros são mais complexos, mas permitem a captura de mais informação relativa às chamadas ao sistema de ficheiros e oferecem mais funcionalidades para a filtragem dos eventos.

Relativamente ao sistema desenvolvido, este possibilita encapsular os mecanismos de recolha de eventos, fornecendo assim uma interface comum de monitorização e oferece funcionalidades adicionais de filtragem de eventos que simplificam o seu processamento.

2.4 Sistemas de disseminação de eventos

Um sistema de disseminação de eventos [10, 16] pretende fornecer um mecanismo de partilha de informação, sob a forma de eventos, entre participantes de um determinado grupo, assegurando que cada um deles recebe a informação que lhe interessa de forma eficiente e consistente. Usando um sistema deste tipo, os emissores e os receptores não necessitam de se preocupar com a implementação da comunicação.

Uma aproximação muito usada em larga escala é o modelo *publish/subscribe*. Neste modelo, os produtores publicam os eventos que o sistema dissemina pelos subscritores, bastando para tal que estes indiquem os seus interesses. Este modelo é genérico e pode ser implementado

segundo uma arquitectura cliente/servidor ou *peer-to-peer*.

Nos sistemas baseados em eventos, os objectos e as acções são modelados de modo a serem transmitidos sobre a forma de notificações. O tipo de eventos que são disseminados pode variar, sendo possível a criação de filtros de eventos sobre canais de disseminação de eventos. Quando não existem filtros, os subscritores de um canal vão receber todos os eventos que são enviados para esse canal. Os filtros permitem que se escolham anúncios específicos a receber, com base na informação que transportam (analisando-se o seu conteúdo).

Existem muitos sistemas de disseminação de notificações, que usam diversas abordagens na propagação das mensagens, como por exemplo: *flooding*, *multicast*, *routing table* usando o princípio *learning by reverse path* ou com base em chaves. Em [10, 16] apresentam-se *surveys* recentes sobre sistemas de disseminação. Nesta secção apresentam-se em detalhe dois sistemas de disseminação de eventos, sendo o segundo especificamente desenvolvido para ambientes móveis.

2.4.1 Hermes

O Hermes [23] é um sistema de *middleware* de disseminação de eventos que se pretende que seja escalável e suficientemente robusto para permitir o seu uso em qualquer tipo de aplicação que necessite de fazer uma interacção baseada em eventos.

Ao contrário de outros sistemas, o Hermes fornece controlo de acessos, fiabilidade e uma boa capacidade de integração com linguagens orientadas a objectos.

Durante a fase de desenho do sistema, foram definidos cinco requisitos fundamentais para o Hermes, os quais são importantes para qualquer sistema de disseminação de eventos:

- **Escalabilidade**: O sistema deve suportar um largo número de utilizadores, o que faz com que a arquitectura do sistema deva ser distribuída, para evitar *bottlenecks*. Também não deve ser mantido em qualquer nó o estado completo do sistema e os seus recursos;

- **Interoperabilidade**: É importante garantir que mesmo com a heterogeneidade normalmente associada a um elevado número de máquinas, estas continuem a conseguir comunicar e interagir entre si independentemente do seu ambiente (por exemplo, sistemas operativos diferentes). Para isso o sistema deve ser independente de qualquer plataforma ou suporte particular;

- **Fiabilidade**: Diferentes clientes necessitam de diferentes tipos de qualidade de serviços e o sistema deve suportar vários contextos, adequados à aplicação que o utiliza. Também têm que ser implementados métodos de tolerância a falhas e replicação para garantir a fiabilidade do sistema;
- **Expressividade**: Aplicações distribuídas necessitam de poder exprimir os seus interesses, subscrevendo eventos particulares ou grupos de eventos com determinadas características.
- **Usabilidade**: é importante que o sistema seja simples de integrar em qualquer aplicação que o queira utilizar.

Relativamente à arquitectura do sistema, existem duas componentes distintas: os *event clients* e os *event brokers*. Os *event clients* podem fazer anúncios ou subscrever eventos e usam os serviços fornecidos pelo *middlewar*e para comunicar entre si, utilizando eventos. Os *event brokers* representam o próprio *middleware* e fornecem uma implementação distribuída dos serviços utilizados pelos clientes. Uma visão da organização dos *event clients* e dos *event brokers* é dada na Figura 2.7.

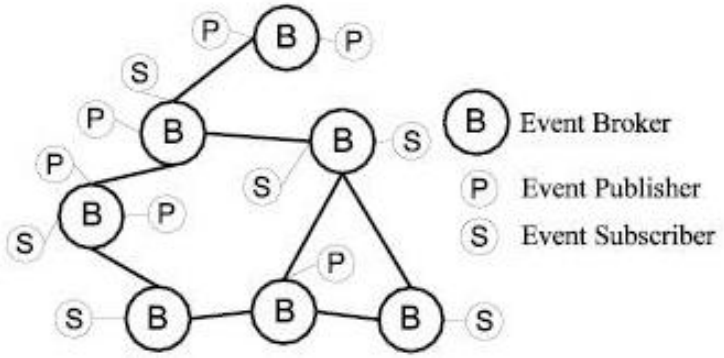

Figura 2.7 Ligação entre os dois tipos de componentes do sistema Hermes (figura retirada de [23]).

Todas as funcionalidades de *middleware* são implementadas nos *brokers* o que torna os clientes facilmente integráveis em qualquer aplicação. Os clientes ligam-se aos *brokers* para acederem aos seus serviços e os *brokers* estão ligados entre si para trocarem informação. Os *brokers* aceitam pedidos de anúncio e de subscrição e quando existe uma nova notificação, estes recebem-na e difundem-na para os outros *brokers* existentes no sistema, conseguindo-se assim fazer chegar a informação a todos os subscritores interessados.

Os nós *brokers* organizam-se segundo uma rede sobreposta (semelhante ao Pastry [24]). Desta forma, cada nó conhece apenas alguns outros e para enviar uma mensagem para um destinatário encaminha-a para outro nó até que ela chegue ao destino.

Os clientes que fazem os anúncios registam-se no sistema de forma a criar uma árvore de disseminação de eventos. Os clientes que fazem as subscrições precisam de entrar nesta árvore para que os eventos lhes cheguem.

O Hermes usa um sistema de ponto de encontro, ou seja, para cada anúncio existe um nó na rede que o guarda. Os clientes que querem subscrever esse anúncio comunicam com o nó responsável, indicando que se pretendem juntar à árvore de disseminação de eventos.

Este sistema suporta a possibilidade de definir tipos de eventos que são usados para distinguir os eventos gerados e permitir que os subscritores recebam apenas os eventos que lhes interessam.

Um subscritor pode receber todos os eventos relacionados com um certo tópico ou apenas com um conteúdo específico. Para tal, é possível criar subtipos de eventos dentro de cada tópico e definir expressões regulares que podem ser analisadas em tempo de execução por parte dos *brokers* para fazer a filtragem dos eventos que um subscritor pretende receber.

Apesar de o Hermes ser um sistema de disseminação de eventos genérico, a sua implementação apresenta algumas diferenças relativamente ao sistema proposto nesta dissertação.

Em primeiro lugar, na organização dos nós, enquanto no Hermes é feita uma distinção entre nós de *publish* e *subscribe*, no Forby todos os nós são simultaneamente emissores e receptores, criando-se uma rede com todos os nós de um grupo. Também não existe a necessidade de manter os nós especiais (*brokers*) para disseminar informação, o que simplifica a infra-estrutura a utilizar e permite que o sistema funcione quando os nós falhem, ao contrário do Hermes que necessita dos *brokers* para manter a disseminação.

A não existência destes nós especiais também evita que se enviem mensagens para destinatários que não pretendem recebê-las (por exemplo, pode haver mensagens enviadas entre *brokers* que não têm clientes interessados nelas ligados, mas que servem de ponte para outros *brokers*).

2.4.2 Elvin com suporte para ambientes móveis

Em [26], os autores propõem uma extensão ao sistema Elvin [9] para que este ofereça suporte para ambientes móveis. O Elvin, que já tinha sido referido na secção 2.2.4, é um

sistema de disseminação de eventos baseado no modelo *publish/subscribe*.

O Elvin utiliza um ou mais servidores, agregados em federações, para fazer o encaminhamento dos eventos entre os clientes. O sistema considera como cliente qualquer utilizador que pretenda disseminar ou receber eventos. Os eventos são analisados e filtrados no servidor, o que permite que cada cliente defina o tipo de evento que quer receber.

O sistema de suporte para clientes móveis passa pela criação de uma *proxy* que é vista pelo servidor como um cliente normal, e pelos clientes como o servidor. Esta *proxy* encapsula diversos mecanismos que permitem fornecer o suporte para ambientes móveis, sem ser necessário fazer modificações no sistema Elvin.

Um primeiro mecanismo incorporado na proxy é o suporte para a persistência de eventos. Neste contexto, a *proxy* funciona como um repositório de eventos. Como existe a possibilidade de serem disseminados eventos quando os clientes estão desconectados é necessário armazená-los, uma vez que o Elvin não tem essa funcionalidade. Quando os clientes se voltam a ligar ao sistema, ligam-se à *proxy* e esta entrega-lhes os eventos gerados durante o período de desconexão.

Como a *proxy* tem que lidar com diversas subscrições de vários clientes, foi introduzido no sistema o conceito de sessão, que faz um mapeamento entre cada cliente e o conjunto de eventos que pretende receber.

Uma vez que é comum cada utilizador ter vários dispositivos fixos e móveis, é razoável assumir que este pode decidir em qual dos dispositivos quer receber as notificações, pelo que uma sessão deve ter em conta todo o grupo de dispositivos do utilizador.

Com esta diversidade de dispositivos, é possível que os utilizadores queiram receber as notificações num determinado dispositivo, mas mais tarde receber essa notificação noutro. A *proxy* encarrega-se de fazer essa gestão, garantindo que as notificações não são entregues duas vezes ao mesmo cliente, excepto se isso for explicitamente pedido.

Outro aspecto importante é a definição de um *time-to-live* para cada notificação gerada. Como se trata de utilização sobre um ambiente móvel, é possível que um cliente permaneça desconectado durante um longo período temporal. Ao voltar a ligar-se ao sistema é provável que muitas das notificações entretanto geradas já não sejam interessantes, pelo que devem ser removidas do repositório em vez de serem entregues ao cliente.

A implementação teve que ter em conta a forma como o sistema Elvin foi desenhado. Em primeiro lugar, a *proxy* procura um servidor do Elvin para estabelecer a ligação. De seguida, espera pelas ligações dos clientes e faz o processamento dos anúncios e subscrições, que podem

ser locais à *proxy* ou terem que ser encaminhados para o servidor.

Quando a *proxy* recebe uma notificação, é feita uma verificação das suas subscrições e entrega-a no caso de existir uma sessão aberta. Caso contrário guarda-a no repositório para posteriormente serem enviadas ou consultadas pelos clientes.

Comparativamente com o Forby, o Elvin com suporte para desconexão mantém algumas das diferenças já apresentadas no sistema Hermes, como a necessidade de manter nós especiais (servidores) para fazer a disseminação ou a distinção entre emissores e receptores.

Comparando o suporte para desconexão oferecido para os dois sistemas, considera-se que o definido para o Forby é mais flexível e robusto, dado que não se baseia na introdução de mais um participante especial do sistema que, em caso de falha, limita todo o suporte que se pretende oferecer.

A substituição da *proxy* por um sistema de repositório e *caching* aumenta a resiliência do sistema, garantindo que em caso de falha de alguns nós, seja possível obter a informação pretendida. Também não é necessário manter um nó ligado à rede a todo o momento para fazer o *log* da informação.

No entanto, e caso a informação gerada deva estar disponível em qualquer momento, para além da utilização de um repositório remoto é possível criar, de forma simples, um nó especial que se liga aos restantes elementos do grupo e armazena as mensagens. Este nó não será mais do que uma aplicação que, ao receber um evento, o armazena e o disponibiliza a pedido.

3 . Desenho do sistema

Neste capítulo apresenta-se o desenho do sistema desenvolvido para a disseminação de notificações sobre ficheiros partilhados. Em primeiro lugar, são definidos os objectivos e requisitos que foram definidos para o sistema. De seguida, apresentam-se as principais funcionalidades oferecidas e são analisados os componentes em que este se divide.

3.1 Objectivos e Requisitos

Para o desenho e implementação do Forby, foram definidos alguns objectivos e requisitos que guiaram o trabalho apresentado. Nesta secção é feita uma listagem dos mesmos.

Os objectivos e requisitos genéricos incluem ter um sistema eficiente, genérico e facilmente utilizável. Com esses objectivos em mente, é possível definir outros mais concretos no que diz respeito à recolha de eventos e à sua disseminação.

Para ter um sistema genérico, utilizável em múltiplas plataformas, a recolha de eventos deve ser modular e transparente para a aplicação. Isto permite que, para diferentes sistemas, se criem diferentes módulos de captura de informação acerca do acesso ao sistema de ficheiros, sem alterar o funcionamento da aplicação.

Uma vez que se pretendem obter eventos sobre acessos ao sistema de ficheiros, a área de trabalho a monitorizar deve ser definida pela aplicação, bem como os tipos de eventos que se pretendem recolher, dado que os eventos interessantes variam de aplicação para aplicação. Por exemplo, uma aplicação de disseminação de alterações está interessada em escritas, enquanto uma aplicação que fornece de informação de *awareness* está interessada nas leituras (para notificar os utilizadores do estado das restantes réplicas). Para isso, a aplicação deve definir filtros (seguindo a interface do sistema) que definem os eventos que lhe são entregues.

No que diz respeito à disseminação da informação recolhida, é importante que o sistema seja eficiente. A rede deve estar organizada de tal forma que permita fazer uma distribuição de carga entre os nós. Isso levou a que o sistema seguisse um modelo *peer-to-peer*, onde todos os participantes do grupo cooperam na disseminação da informação.

É também importante tentar minimizar a informação que passa pela rede. Assim, o sistema deve garantir que as notificações circulam apenas entre os nós interessados e que as notificações obsoletas não são propagadas. Neste sentido, a rede está organizada por grupos, fazendo com

que a informação circule apenas dentro de um determinado grupo que pretende receber os eventos. Para minimizar a informação enviada, e tendo em conta a semântica da informação (acesso a ficheiros locais), foram definidas propriedades nos eventos que permitem ao sistema eliminar os eventos obsoletos. Por exemplo, se ocorrerem várias modificações sucessivas num mesmo ficheiro, basta manter e propagar a informação da última modificação, em vez de enviar todas as alterações.

O sistema deve também permitir à aplicação utilizar todos os recursos existentes ao seu dispor. Deste modo, o Forby, caso a aplicação assim o defina, pode utilizar um servidor central para armazenar informação ou como ponto de encontro. No caso de não existir, o sistema deve continuar a funcionar, organizando-se então os nós numa DHT [24, 2].

Deve ser possível a um nó entrar na rede de disseminação mesmo estando por trás de uma *firewall*, sem necessidade de configuração por parte do administrador de rede. Para isto os nós públicos podem servir de ponte entre várias redes privadas. Quando não existem nós públicos, o sistema pode funcionar usando um repositório remoto para propagar a informação. Esta propriedade é importante para permitir que pequenos grupos de utilizadores consigam pertencer à mesma rede de disseminação, com um suporte mínimo da infra-estrutura, o qual é obtido pela utilização do outro serviço (e.g. *email*).

A aplicação deve decidir se armazena a informação dos eventos gerados localmente ou se deve usar um repositório remoto (servidor central). Este aspecto influencia a forma como são feitos os pedidos de eventos que não foram recebidos. Ao utilizar um repositório remoto, os eventos estão acessíveis a qualquer momento por qualquer participante, possibilitando a troca de notificações entre redes privadas, sem necessidade de se recorrer a um nó público que faça a ponte entre essas redes. Com um repositório local, os pedidos devem ser encaminhados pela rede de participantes, até se encontrar a resposta a esses pedidos.

O sistema deve incluir um mecanismo de tolerância a falhas, permitindo que um nó que esteve desconectado durante um período limitado possa aceder a todos os eventos. Para tal, pode combinar-se uma política de *best effort* no que diz respeito à entrega de mensagens com a recuperação de mensagens não recebidas (por falhas na rede ou desconexão).

É importante realçar que o desenho do sistema tem em conta que o conjunto de utilizadores que faz parte de um grupo é pequeno e que o número de eventos é reduzido.

3.2 Arquitectura

O sistema de disseminação de notificações sobre ficheiros partilhados fornece à aplicação a possibilidade de capturar eventos do sistema de ficheiros e de os encaminhar para todos os elementos de um grupo. Para tal, existem dois componentes distintos: o sistema de detecção e o sistema de disseminação.

O sistema de detecção é responsável por capturar os eventos resultantes do acesso ao sistema de ficheiros. A aplicação deve definir quais os ficheiros a monitorizar e os tipos de eventos que pretende recolher para cada ficheiro. Com essa informação, os eventos são filtrados e apenas são entregues à aplicação se forem interessantes. A aplicação processa estes eventos e encaminha-os para o sistema de disseminação, que gera um novo tipo de evento (que encapsula o recolhido pelo sistema de ficheiros ou gerado pelas aplicações) e o propaga para os outros nós do sistema.

O papel do sistema de disseminação é permitir que todos os participantes de um grupo se interliguem entre si e troquem informação sobre os eventos gerados em cada nó. Este sistema utiliza, numa primeira fase, uma política *best effort*, ou seja, não garante que as mensagens sejam entregues no momento em que são geradas. Adicionalmente, existe um mecanismo de recuperação de falhas que permite requisitar, a qualquer momento, os eventos não recebidos, para que cada nó tenha uma visão completa do estado do sistema, mesmo que não receba as notificações quando elas são propagadas.

Na Figura 3.1 apresenta-se o desenho das componentes do sistema e a sua interacção com a aplicação, com o sistema de ficheiros e com a rede.

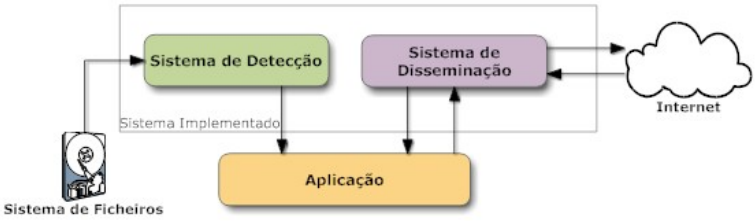

Figura 3.1 Componentes do Forby.

De seguida apresentam-se mais detalhadamente as características de cada um dos componentes do sistema.

3.2.1 Sistema de detecção

O sistema de detecção é responsável por recolher e filtrar todos os eventos resultantes do acesso aos ficheiros monitorizados. Para isso, este componente encontra-se dividido em três módulos (Figura 3.2): Event Receiver, Event Filter e Event Manager.

O Event Receiver faz a monitorização do sistema de ficheiros e produz eventos relativos a acessos aos ficheiros. Os eventos produzidos podem ser do tipo *open*, *close*, *read*, *write*, *create* ou *delete*.

Este módulo abstrai uma implementação específica, conseguindo-se deste modo que existam diferentes formas de fazer a monitorização do sistema de ficheiros, garantindo-se portabilidade para várias plataformas.

O Event Filter representa um filtro definido pela aplicação e permite que os eventos gerados e recolhidos pelo Event Receiver sejam filtrados antes de serem processados pela aplicação. Este filtro define os ficheiros que devem ser monitorizados, que eventos são importantes para cada ficheiro e faz a validação dos eventos a entregar à aplicação.

O Event Manager é responsável por receber um evento recolhido pelo Event Receiver, filtrá-lo de acordo com o Event Filter definido pela aplicação e no caso de ser um evento interessante, entregá-lo à aplicação para ser processado. Este módulo serve então como ponto de ligação entre as aplicações e o sistema de recolha e filtragem de eventos.

Os eventos obtidos neste módulo são processados assincronamente, ou seja, a chamada ao sistema que gera o evento não bloqueia à espera que este seja filtrado e entregue à aplicação.

A Figura 3.2 mostra a forma como os módulos se interligam entre si e qual a ordem lógica seguida por um evento que entra no sistema.

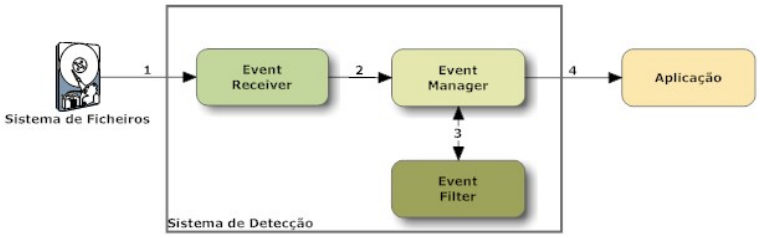

Figura 3.2 Organização dos módulos do sistema de detecção.

3.2.2 Sistema de disseminação

O sistema de disseminação possibilita a uma aplicação entrar numa rede de nós pertencentes a um grupo para trocar mensagens. Apesar de ser um sistema de disseminação genérico, foi desenhado tendo em conta os requisitos específicos de um sistema para disseminação de eventos sobre modificações de ficheiros entre grupos de utilizadores.

A disseminação é feita segundo um modelo *publish/subcribe*, onde cada nó é simultaneamente emissor e receptor de eventos, ou seja, todos os nós de um grupo podem enviar mensagens para os restantes nós. Ao contrário de outros sistemas de disseminação de eventos, não é necessário distinguir entre os nós que publicam mensagens e nós que as subscrevem.

A arquitectura do sistema baseia-se num modelo *peer-to-peer*. Cada nó, ao entrar no sistema, liga-se a outro nó formando-se assim uma árvore de disseminação entre todos os participantes, como se detalha na secção 4.3.

Relativamente à arquitectura utilizada para a obtenção da lista de participantes de um grupo (para entrar na rede), o sistema permite que seja a aplicação a definir o que mais se adequa às suas necessidades, sendo esta uma característica importante do sistema de disseminação. Em concreto, a aplicação pode optar por utilizar um servidor central como o SVN/CVS, um servidor de email que use IMAP, ou outro por si implementado ou então recorrer a uma DHT (Pastry) como ponto de encontro para a obtenção dos participantes activos do grupo. Todos os nós participam na manutenção da lista de participantes como se detalha na secção 4.3.

Cada nó mantém localmente uma cache das notificações que recebe. Existe também um repositório que armazena os eventos gerados localmente em cada um dos nós. Este repositório pode ser local a cada nó ou pode ser criado num servidor remoto, como o SVN/CVS ou o *Gmail*, caso esse recurso se encontre disponível e a aplicação o pretenda.

Sendo a política de propagação das mensagens, numa primeira fase, do tipo *best effort*, é possível que nem todos os participantes recebam os eventos no momento em que são disseminados, por exemplo, por haver uma quebra momentânea na rede de disseminação ou por estarem desconectados do sistema.

Para permitir que um nó consiga ver todas as mensagens válidas que não recebeu (criando-se assim um mecanismo simples para fornecer fiabilidade e suportar desconexão), este pode requisitar estas mensagens a outro nó (usando uma aproximação de propagação epidémica [5]). Um nó, ao receber o pedido, verifica se na sua cache existem as mensagens pretendidas e em caso afirmativo responde com essas mensagens. No caso de se utilizar um repositório remoto

de eventos, os pedidos podem ser feitos a esse repositório, sem necessidade de haver troca de mensagens com os restantes participantes do sistema.

Outro aspecto importante e que diferencia o sistema apresentado dos restantes é a possibilidade de utilizar as propriedades dos eventos do sistema de ficheiros para tornar a disseminação mais eficiente. Dado que se tratam de eventos que reflectem os acessos ao sistema de ficheiros, os eventos mais recentes podem tornar obsoletos os eventos anteriores. Por exemplo, uma escrita num ficheiro torna obsoletas as escritas anteriores. Nestes casos, a visão que a aplicação tem do estado do sistema de ficheiros é a mesma se receber todas as mensagens ou apenas a última. Desta forma é possível minimizar os dados que são enviados pela rede, conseguindo-se uma melhor eficiência nos dados transferidos.

4. Implementação

Com base no desenho apresentado no capítulo 3, foi implementado um protótipo do Forby. Este capítulo apresenta a descrição da implementação de cada um dos componentes do sistema. Esta descrição não será exaustiva, procurando-se focar os aspectos que se consideram mais importantes.

4.1 Ambiente

O protótipo foi desenvolvido na linguagem Java, em ambiente Windows e Linux. A escolha da linguagem deveu-se, em primeiro lugar, à sua portabilidade, tornando possível a utilização do sistema em múltiplas plataformas sem necessidade de alteração de código. Também o conjunto de classes fornecidas pela linguagem, principalmente ao nível de comunicação, facilitou o desenvolvimento do protótipo.

O desenvolvimento em ambiente Windows foi motivado pelo inexistente suporte fornecido pelo sistema VC^2 para este ambiente, no que diz respeito à recolha de eventos do sistema de ficheiros. Em Linux, utilizou-se o suporte já existente para a recolha de eventos feito para o VC^2.

4.2 Sistema de detecção

Como foi apresentado na secção 3.2.1, o sistema de detecção encontra-se dividido em três módulos: Event Receiver, Event Filter e Event Manager. A implementação de cada um deles é analisada de seguida.

4.2.1 Event Receiver

O módulo Event Receiver é responsável pela monitorização e captura de eventos do sistema de ficheiros. Esses eventos, denominados no Forby como FSEvents, são posteriormente entregues ao Event Manager para serem filtrados e enviados para a aplicação.

Um FSEvent (Figura 4.1) agrupa a informação importante sobre um evento do sistema de

ficheiros. Em geral, guarda o tipo de evento, o ficheiro associado a esse evento e a data de criação. Este FSEvent pode ser estendido, para conter mais informação, de acordo com as necessidades das aplicações que usem este sistema.

```
// Variáveis da classe:

     private int type;
     private String file;
     private Date date;

// Métodos da classe:

     // Devolve o tipo de evento
     public int getType();

     // Devolve o nome do ficheiro acedido
     public String getFile();

     // Devolve a data de criação do evento
     public Date getDate();

     // Devolve a data de criação do evento formatada
     public String getTime();

     // Indica se um evento comuta com outro
     public boolean commute(FSEvent other);

     // Indica se um evento torna outro obsoleto
     public boolean overwrite(FSEvent other);
```

Figura 4.1 Interface de um FSEvent.

A forma como a monitorização e recolha de eventos é feita difere de plataforma para plataforma. Assim, para cada ambiente de utilização deve haver uma implementação apropriada, caso contrário a portabilidade do sistema seria posta em causa. A única restrição que uma implementação do módulo deve ter passa por assegurar que exista um apontador para o Event Manager, para que seja possível reencaminhar os FSEvents capturados.

Neste protótipo foram implementadas duas variantes deste módulo. A primeira usa o JNotify [1] que utiliza uma API de monitorização bem definida, simples de utilizar e que funciona tanto em Windows como em Linux. A segunda aproximação é mais genérica, recebendo os eventos através de um socket UDP.

A implementação utilizando o JNotify necessita apenas de um caminho para a área a monitorar. Os parâmetros da monitorização consistem no caminho e no tipo de eventos que se

pretendem recolher (e.g. a criação ou modificação de um ficheiro). Quando um evento correspondente ao caminho, com os atributos definidos é gerado, a biblioteca do JNotify notifica o módulo (seguindo um mecanismo de *callback*) e passa-lhe o evento, indicando o ficheiro que foi acedido e o tipo de acesso. Com esta informação é então gerado um FSEvent que é encaminhado para o EventManager.

Na Figura 4.2, pode ver-se um esquema da implementação do módulo utilizando o JNotify.

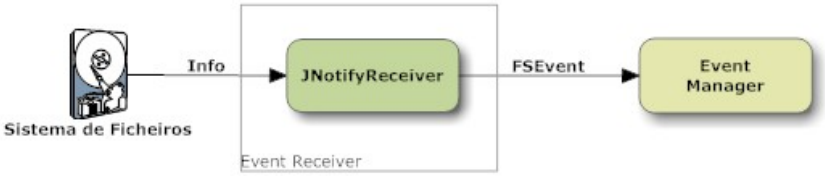

Figura 4.2 Módulo de detecção implementado com JNotify.

A segunda aproximação, utilizando um *socket* UDP, é genérica, permitindo a integração de qualquer sistema de monitorização, sendo apenas necessário que esse sistema envie uma mensagem com a informação pretendida.

Esta implementação consiste em ter um *socket* UDP activo que vai recebendo a informação recolhida por uma aplicação externa. É, no entanto, necessário definir uma forma de interpretar esses dados. Para isso, foi definida uma interface denominada IEventParser, cuja implementação deve gerar um FSEvent baseado no conteúdo da mensagem recebida pelo *socket* (Figura 4.3).

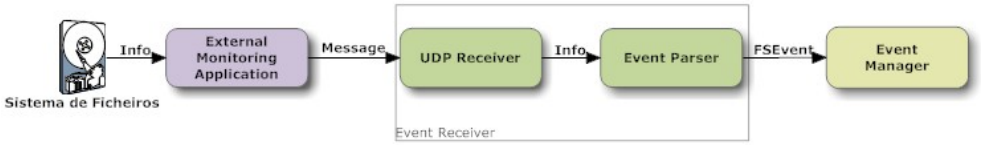

Figura 4.3 Módulo de detecção implementado com *socket* UDP.

4.2.1.1 Aplicações de monitorização externas e Event Parsers associados

O utilizador/programador pode optar por utilizar uma aplicação externa para fazer a monitorização do sistema de ficheiros, permitindo que o Forby seja utilizado em qualquer plataforma. Esta opção de implementação possibilita ainda que, no futuro, se integrem mecanismos de captura de eventos diferentes dos actuais, sem necessidade de alterar o protótipo apresentado.

Existem diversas ferramentas que podem ser utilizadas para capturar os eventos, como por exemplo, a API Installable File Systems [18] ou o sistema Dokan [7] para Windows e o Fuse [31] ou o Fist [34] para Linux. Para este protótipo, como exemplo, foi utilizada a API Installable File Systems para implementar uma aplicação de monitorização externa.

Como referido no capítulo 2, utilizando a API Installable File Systems é possível desenvolver um *minifilter* driver que intersecte as chamadas ao sistema de ficheiros.

O *minifilter* implementado intersecta todas as chamadas ao sistema de ficheiros (seguindo o apresentado na secção 2.3.1) e filtra as que devem ser enviadas para o Event Receiver definido pela aplicação, com base no código de operação que indica o tipo de chamada e no nome do ficheiro sobre o qual se está a trabalhar.

Este filtro apresenta algumas limitações no que diz respeito à captura de informação do sistema de ficheiros. Ao contrário do Linux, em Windows não é possível obter de forma fiável algumas das chamadas interessantes como um *file open* ou um *file close* (uma vez que nem sempre são geradas/capturadas).

O Event Parser associado a este *minifilter* faz o mapeamento entre o código de operação resultante do driver no tipo de evento do FSEvent. Como exemplo, quando o driver intercepta uma escrita num ficheiro, gera um código IRP_MJ_WRITE, envia uma mensagem com essa informação e o Event Parser gera um novo FSEvent cujo tipo é FILE_WRITE. Este inclui o nome recebido e a data actual do sistema e, caso seja necessário, outros parâmetros importantes para o funcionamento da aplicação em questão.

4.2.2 Event Filter

O módulo Event Filter permite que uma aplicação defina os eventos que quer receber. O programador deve então desenvolver o filtro implementando uma interface específica, apresentada na Figura 4.4, cujos métodos permitem ao Event Manager validar um evento e entregá-lo à aplicação.

```
public void addFiles(String path);
public void removeFile(String file);
public boolean validateEvent(FSEvent e);
public FSEvent deliver(FSEvent e);
```

Figura 4.4 Interface do EventFilter

Os dois primeiros métodos permitem indicar quais os ficheiros que devem ser filtrados. Quando uma aplicação cria um filtro, deve adicionar os ficheiros que pretende que esse filtro processe. Isso é feito recorrendo ao método *addFiles()*. Este método, assim como o *removeFile()*, pode também ser invocado durante o processamento de um evento, caso este leve a que seja necessário adicionar ou eliminar ficheiros do filtro, por exemplo, por ser gerado um evento de criação ou remoção de ficheiros na área de monitorização.

O método *validadeEvent()* é usado para, dado um FSEvent recolhido, validar se este deve ou não ser processado. Por fim, o método *deliver()* processa o evento, indicando se este deve ou não ser enviado para a aplicação.

Este último método permite que certos eventos, apesar de serem interessantes para a aplicação, não sejam enviados, podendo ser logo processados no filtro. Como exemplo, se um ficheiro for criado e a aplicação não necessitar de propagar essa informação para os restantes elementos do grupo, mas se o quiser adicionar ao filtro, pode fazê-lo directamente no EventFilter, no método *deliver()*.

Outro exemplo de um processamento que se pode efectuar neste método é a filtragem de sequências de eventos, como *open - write - write - close*. Esta combinação de eventos permite que uma aplicação, que apenas necessite de propagar a informação sobre a alteração de um ficheiro, apenas receba o último evento do tipo *write*, evitando-se assim a disseminação de notificações desnecessárias.

4.2.3 Event Manager

O Event Manager faz a gestão dos eventos do sistema de ficheiro, fazendo a ligação entre o Event Receiver, o Event Filter e a aplicação. A implementação deste módulo passa por fornecer à aplicação um método de subscrição no Event Manager e outro método que permita a um Event Receiver encaminhar os FSEvents recolhidos.

A subscrição é feita através do registo da aplicação no Event Manager, utilizando um mecanismo de *callback* que permite que o Event Manager invoque uma função definida na aplicação, para fazer encaminhamento dos eventos interessantes. Nesta subscrição é também enviado o Event Filter que permite fazer processamento do evento, de acordo com os interesses da aplicação.

Na Figura 4.5 encontra-se esquematizada a interacção entre o Event Manager, a aplicação e os restantes módulos pertencentes ao sistema de detecção.

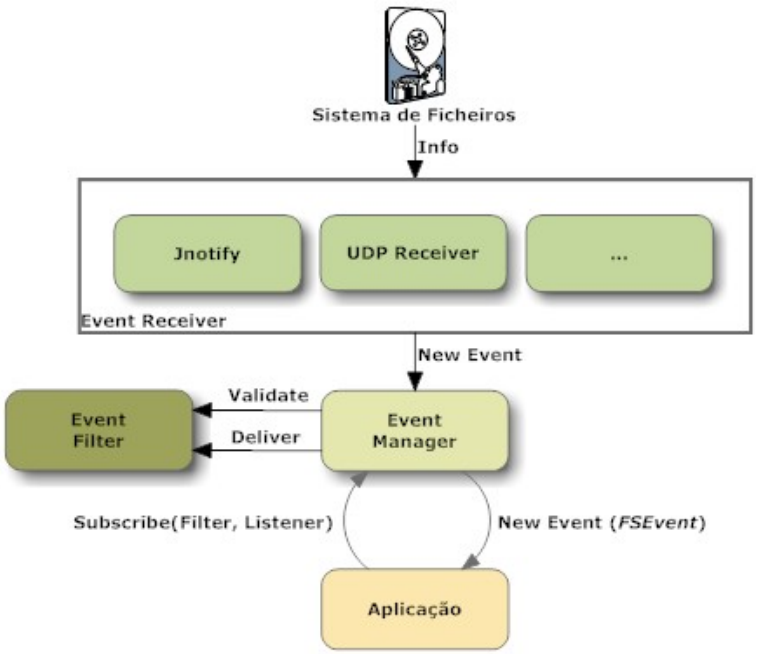

Figura 4.5 Interacção entre Event Manager e aplicação.

4.3 Sistema de disseminação

O sistema de disseminação é responsável pela ligação e troca de informação entre os participantes de um grupo. Existem vários aspectos a ter em conta na implementação deste módulo, que o diferenciam dos outros sistemas de disseminação existentes, como a forma de descoberta de outros participantes activos ou como se minimiza a informação enviada entre os nós de um grupo.

4.3.1 Organização dos participantes

O Forby segue um modelo *publish/subscribe* onde todos os participantes podem enviar ou receber notificações sem necessidade de se registarem especificamente como emissores ou receptores.

Como já foi referido de forma breve, na arquitectura do sistema, os participantes de um grupo ligam-se entre si formando uma árvore de nós. Cada grupo terá a sua própria árvore de disseminação, tornando os grupos independentes entre si.

A árvore de disseminação que é formada tem por base um algoritmo determinista que permite a todos os nós manterem uma visão geral da organização do grupo. Desta forma, quando um nó sai/falha, é propagada uma mensagem e os restantes nós reorganizam-se e estabelecem as ligações necessárias à manutenção da disseminação.

Cada nó mantém então, a cada momento, a lista dos participantes activos do grupo. Esta informação é obtida recorrendo-se a um mecanismo de ponto de encontro. A identificação do grupo é feita utilizando-se um identificador único, que permite a um participante recolher informação sobre esse grupo e juntar-se a ele.

Uma vez que a organização é feita por grupos, consegue-se evitar que as notificações circulem por nós que não estão interessados nelas (membros de outros grupos, por exemplo), conseguindo-se um melhor aproveitamento da largura de banda.

Também não existe a necessidade de criar nós especiais na rede, para interligar os participantes e encaminhar as notificações entre eles, minimizando o nível de recursos que é necessário manter para o funcionamento do sistema.

4.3.2 Sistema de ponto de encontro

Para entrar num grupo, um participante tem que determinar que outros participantes se encontram activos, para que se estabeleçam as ligações necessárias à entrada na árvore de disseminação.

Essa informação é disponibilizada através de um sistema de ponto de encontro, onde todos os nós de um grupo se registam. Este mecanismo permite a um nó, no momento da entrada, requisitar a lista de nós que constituem o grupo.

A arquitectura modular do sistema de disseminação permite que se utilizem diferentes tipos de ponto de encontro. Neste protótipo é disponibilizado suporte para a utilização de um servidor central (CVS, SVN ou *Gmail*) ou de uma DHT (Pastry) como ponto de encontro. A utilização dos diferentes tipos de ponto de encontro é feita de forma transparente para o resto do sistema, sendo apenas necessário que estes retornem a lista de participantes activos de um grupo.

A escolha do tipo de ponto de encontro a utilizar é feita pela aplicação, tendo em conta as suas características e os recursos que tem disponíveis.

4.3.2.1 Servidor Central

No caso de existir um servidor disponível, este pode ser utilizado como ponto de encontro, sem ser necessário mudar a sua arquitectura ou implementação. A informação sobre os participantes é guardada num ficheiro que é acedido e actualizado por todos os nós do sistema.

Tomando como exemplo o SVN, implementado no protótipo, é criado um ficheiro *peers.dat* que é adicionado ao projecto e que mantém a informação dos participantes activos. Quando um novo participante pretende juntar-se à rede de disseminação, faz o *checkout* desse ficheiro, recolhendo a informação dos restantes elementos do grupo. Depois de tomar o seu lugar na rede, com base na análise do ficheiro recebido, actualiza o ficheiro no servidor, indicando que se encontra activo.

No caso do *Gmail*, quando um participante pretende entrar na rede de disseminação deve obter a mensagem do servidor que corresponde aos participantes activos do grupo, utilizando para isso o tópico da mensagem (por exemplo, *subject: groupPeers*). Depois de processar essa informação, é necessário criar uma nova mensagem, adicionando a indicação de que se encontra activo e substituir a mensagem anterior.

Na Figura 4.6 encontra-se um esquema com os passos seguidos na recolha da informação sobre os participantes activos de um grupo, recorrendo a um servidor central.

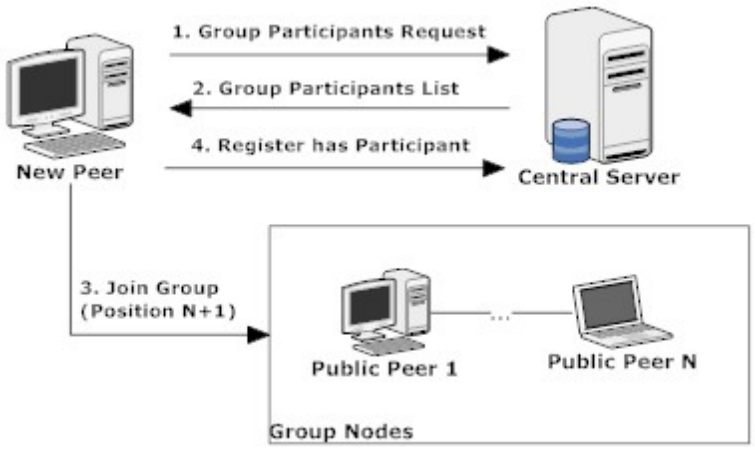

Figura 4.6 Servidor central como ponto de encontro.

4.3.2.2 Rede sobreposta estruturada (Pastry)

Para que o sistema funcione mesmo quando não existe um servidor de suporte (ou quando esse servidor não consegue dar resposta a todos os pedidos) foi implementada uma solução que se baseia numa rede sobreposta estruturada, neste caso, o Pastry [24]. A ideia é organizar todos os participantes de um ou mais grupos segundo uma DHT e usar as suas propriedades para facilitar o encontro entre todos os membros do grupo.

Cada grupo vai ter um nó responsável (*rendez-vous*) que vai agrupar a informação dos participantes activos. O responsável é encontrado utilizando uma função de síntese do nome do grupo para gerar um código *hash*, que vai corresponder a um dos elementos da DHT.

Para um participante se juntar a um grupo, em primeiro lugar, deve tomar o seu lugar na DHT, recorrendo a um endereço de *bootstrap*. De seguida deve contactar o *rendez-vous*, pedindo a lista dos participantes activos e registando-se como participante do grupo.

Como forma de facilitar a entrada de nós na DHT, é utilizado um sistema de *caching* que guarda *n* nós conhecidos. Assim, caso o nó de *bootstrap* não esteja activo numa futura tentativa de conexão, é possível utilizar a *cache* para tentar usar outros nós como ponto de entrada na DHT.

Apesar de ser possível que vários grupos cooperem entre si para facilitar a ligação à DHT, as notificações circulam apenas dentro do grupo, dado que é criada uma rede de disseminação separada da rede sobreposta (Figura 4.7).

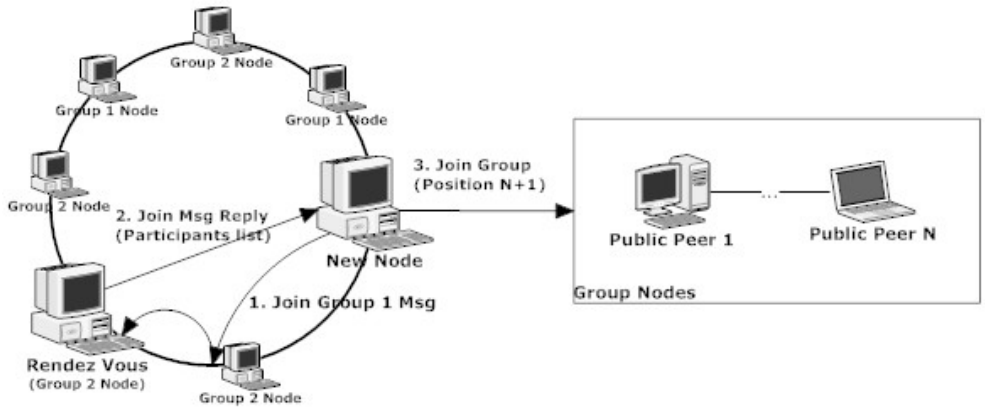

Figura 4.7 DHT como ponto de encontro.

4.3.3 Redes públicas e privadas

As redes privadas, muito comuns nos dias que correm, dificultam a comunicação entre os nós do sistema. Sem a intervenção do administrador da rede, fazendo *portforwarding*, as comunicações dirigidas a um nó de uma rede privada são bloqueadas no *router/firewall*, impossibilitando que se estabeleça uma ligação para troca de notificações.

No geral, as redes privadas permitem que se façam ligações com o exterior e o sistema de disseminação procura tirar partido disso para interligar os participantes de um grupo, sem necessidade de reconfigurar as políticas de comunicação da rede.

Quando um nó entra e detecta que está numa rede privada (analisando o seu endereço de rede), envia uma mensagem *multicast* para saber se existe mais algum nó pertencente ao grupo dentro da sua rede. Se for o primeiro, assume-se como líder do grupo e estabelece uma ligação com um nó público. Se não for o primeiro, o líder do grupo envia-lhe a lista de participantes existentes na rede privada e este, tal como explicado anteriormente, assume a sua posição na árvore de disseminação local.

Tal como acontece com os nós públicos, o líder da rede privada obtém a informação dos participantes activos contactando o ponto de encontro. Com esta informação, o líder pode estabelecer uma ligação com um qualquer nó público activo, entrando assim na árvore de disseminação.

A disseminação das notificações é feita com base na árvore de ligações entre os participantes, onde os nós encaminham as mensagens pela árvore de forma a chegar aos restantes participantes. Os nós públicos enviam a informação para os outros nós públicos e para todos os nós privados com quem têm uma ligação estabelecida. Os nós privados encaminham as notificações que recebem para todos os nós privados da sua rede e o responsável envia-as para o nó público.

Na Figura 4.8 pode ver-se a organização de uma rede de disseminação que inclui duas redes privadas. A ligação entre estas redes é assegurada através dos nós públicos.

4.3.4 Gestão de falhas

O protótipo foi implementado tendo em conta as possíveis falhas de alguns dos nós do sistema. Se um nó da árvore de disseminação falhar, os nós ligados a ele vão notificar o ponto de encontro e disseminar uma mensagem pelo grupo. Esta mensagem leva à reorganização da

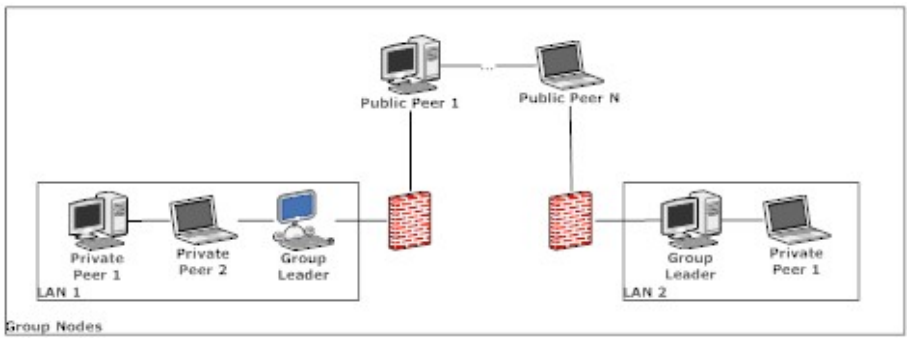

Figura 4.8 Ligação entre nós públicos e privados.

árvore de disseminação, estabelecendo as ligações necessárias para manter a disseminação.

Relativamente às falhas numa rede privada, se o nó público ao qual o líder se encontra ligado falhar, o líder deve procurar outro nó público na sua *cache* e tentar criar uma nova ligação. No caso de falha do líder, um dos nós que está conectado a ele deve assumir o seu lugar, ficando responsável pela rede local.

No caso de se utilizar uma DHT como ponto de encontro, é possível que o *rendez-vous* do grupo falhe. Neste caso, o primeiro nó público da rede de disseminação, que verifica periodicamente se o *rendez-vous* se encontra activo, deve contactar o novo *rendez-vous* e enviar-lhe a sua visão da rede. Desta forma o novo *rendez-vous* pode encaminhar os pedidos de entrada no grupo de acordo com as ligações estabelecidas previamente.

Se o ponto de encontro escolhido for um servidor central, então cabe ao administrador fornecer mecanismos para tolerância a falhas, como por exemplo replicação.

4.3.5 Dissemination Events e as suas propriedades

Quando um FSEvent chega ao sistema de disseminação, proveniente da aplicação que o quer disseminar, é encapsulado noutro tipo de evento: o DisseminationEvent. Este novo evento agrupa toda a informação necessária à propagação do evento pela rede.

Um DisseminationEvent tem um número de sequência, o identificador do nó onde foi gerado (endereço físico e de rede), o FSEvent a propagar e o caminho seguido pelo evento, para o caso da aplicação necessitar de enviar uma mensagem usando o caminho inverso. Para além disso, tem a informação do último evento importante que precede este evento.

Cada nó mantém um repositório de DisseminationEvent, gerados localmente, que pode ser

local ou remoto. Este repositório permite que, a qualquer momento, os nós possam obter a informação dos eventos que foram enviados.

A informação sobre o último evento importante presente no DisseminationEvent baseia-se nas propriedades do FSEvent e permite que se minimize a informação armazenada e que é enviada pela rede, uma vez que um evento pode suprimir outro evento já existente, não sendo necessário então mantê-lo no repositório e enviá-lo para os outros nós.

Quando é recebido um FSEvent para disseminação, é então gerado um DisseminationEvent e é-lhe atribuído um número de sequência com base nos eventos anteriores gerados nesse nó.

De seguida, os eventos presentes no repositório são comparados com o novo evento utilizando as suas propriedades. A comparação indica se o evento anterior comuta ou se suprime o já existente. Se o novo evento suprimir um evento existente, então esse evento é removido do repositório. Se comutarem, o evento é mantido. Depois de comparados os eventos, o campo último evento importante é actualizado para todos os eventos que permaneceram no repositório. Por fim, o novo evento é armazenado.

Quando um evento comuta com outro, a ordem com que são disseminados não tem influência no estado final dos nós. Por outro lado, se um evento suprime outro, então basta apenas ter conhecimento da existência do último evento, dado que o anterior é substituído pelo corrente.

A definição de comutação e supressão de eventos é definida consoante as necessidades da aplicação em questão. Por exemplo, no caso de uma aplicação interessada apenas na última escrita relativa a um ficheiro, um evento do tipo FILE_WRITE suprime os anteriores eventos desse tipo, sobre o mesmo ficheiro, mas outra aplicação pode ter a necessidade de manter todos os eventos de escrita sobre o mesmo ficheiro.

Na Figura 4.9 apresenta-se um exemplo do estado do repositório com a chegada de diferentes eventos e em que um FILE_WRITE suprime os anteriores.

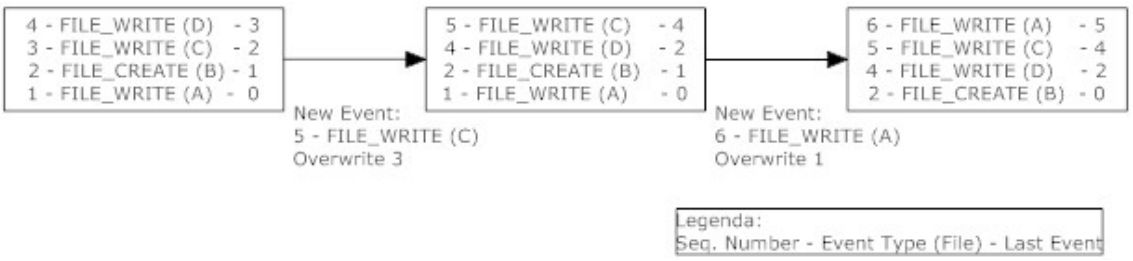

Figura 4.9 Alterações feitas ao repositório.

Relativamente ao tempo de validade dos eventos, o sistema permite que a aplicação defina esse intervalo. Existe uma *thread* que periodicamente verifica se os eventos presentes no repositório são ou não válidos. No caso de não o serem, são eliminados, evitando-se assim que se mantenham eventos que já não são necessários.

4.3.5.1 Repositórios remotos

Outra das funcionalidades oferecidas pelo sistema de disseminação é a possibilidade de armazenar os eventos disseminados num repositório remoto. Esta opção permite que qualquer nó possa, em qualquer altura, aceder a essa informação, mesmo que o elemento do grupo que gerou esse evento não esteja activo ou acessível.

Deste modo, também é possível que participantes do mesmo grupo que se encontrem em redes privadas diferentes consigam comunicar entre si sem ser necessário existir um nó público que faça a ponte entre as duas sub-redes. O líder de cada grupo local contacta o servidor para recolher a informação dos eventos gerados pelos outros participantes e disseminá-la na sua rede local. Esta é outra característica importante do sistema apresentado e que normalmente não é usada noutros sistemas.

Este protótipo utiliza como repositório remoto um servidor SVN ou o *Gmail*. No caso do SVN, quando um DisseminationEvent é gerado, conforme descrito anteriormente, em vez de ser armazenado localmente, é gravado num ficheiro, cujo nome identifica o nó onde o evento é gerado. De seguida estabelece-se uma ligação com o servidor SVN e faz-se *commit* da nova versão do ficheiro no servidor, ficando este disponível para qualquer outro participante fazer *checkout*.

Com o *Gmail*, a ideia é similar à utilização deste servidor como ponto de encontro. Cada participante, ao gerar um evento para disseminar, cria uma mensagem que contém todos os DisseminationEvents desse nó. Essa mensagem é enviada para o servidor e é identificada com o endereço MAC do participante. Para aceder aos eventos disseminados por qualquer nó, basta pesquisar as mensagens presentes no servidor, procurando pelo identificador do emissor do evento.

4.3.6 *Caching*, sincronização e suporte para desconexão

Para além da existência de um repositório de eventos gerados nesse nó, cada nó mantém uma *cache* das notificações que recebe dos outros participantes. O número de notificações mantidas para cada nó pode ser definido pela aplicação.

Uma vez que estamos perante um sistema de notificações *best-effort*, no momento em que as notificações são geradas, não existem garantias que uma notificação emitida chegue a todos os participantes do grupo. As notificações podem perder-se por problemas relacionados com a rede ou por desconexão dos participantes.

O sistema de disseminação deve então fornecer mecanismos para que um nó requisite uma ou mais notificações que não recebeu, para serem processadas. Para determinar quais as notificações perdidas, cada nó mantém um vector de versão [20], onde cada entrada corresponde a um nó do grupo.

A sincronização de versões é desencadeada de duas formas: ao receber uma notificação com um número de sequência diferente do esperado ou quando é feita uma sincronização de vectores de versão (periodicamente).

Quando um nó recebe um evento, é feita uma validação do seu número de sequência. Considerando n como o valor da entrada do vector de versão associado ao nó emissor do evento, temos os seguintes casos:

- Evento com o número de sequência menor ou igual a n: O evento é descartado, uma vez que já foi processado neste nó.
- Evento com o número de sequência igual a $n+1$: O evento é entregue à aplicação, uma vez que é o esperado. O nó corrente conhece todos os eventos anteriores.
- Evento com o número de sequência superior a $n+1$: É necessário adiar a entrega do evento e fazer um processamento adicional.

A análise do último caso é feita com base no último evento importante que está associado ao evento recebido:

- Menor ou igual a n: O evento pode ser entregue à aplicação. Os eventos que foram gerados entre o último evento importante e o recebido foram suprimidos e o nó actual processou o último evento importante.

- Maior que n: É necessário pedir os eventos em falta. O pedido é feito disseminando uma mensagem do tipo EventRequest.

Um EventRequest representa um pedido de eventos e contém a informação sobre o emissor do pedido, a origem dos eventos a pedir e os números de sequência dos eventos em falta.

A mensagem com o EventRequest é enviada para o nó de origem usando o caminho inverso da mensagem recebida e que originou a falha. Ao receber um EventRequest, o nó vai verificar se é a origem dos eventos. Se for o próprio, consulta o seu repositório e gera uma resposta optimizada com os eventos pedidos, tendo em conta aqueles que se encontram válidos (i.e., não foram suprimidos). No caso de ser outro nó, verifica na sua cache se existem alguns dos eventos pedidos e devolve-os. Se com a sua *cache* não conseguir completar totalmente a resposta, então encaminha o EventRequest, retirando dos pedidos os eventos que conseguiu encontrar.

A resposta contendo as notificações pretendidas é enviada directamente para o nó que as requisitou no caso de este ter um endereço público. Se o nó fizer parte de uma rede privada, então a resposta é encaminhada pelos nós do grupo até chegar ao seu destino.

No caso de se utilizar um repositório remoto para armazenar as notificações, os pedidos de eventos podem ser feitos contactando o repositório, no caso de não se obter resposta ao EventRequest enviado.

Se a aplicação utilizar como repositório um servidor SVN, em vez de disseminar um EventRequest pela rede para tentar receber os eventos em falta, pode optar por estabelecer uma ligação com o servidor e pedir o ficheiro com os eventos de um determinado nó. Depois de receber o ficheiro, processa-o, procurando os eventos em falta, para entregar à aplicação. Finalizado este processo, o vector-versão é actualizado e o ficheiro local apagado.

Esta aproximação reduz o número de mensagens enviadas pelo grupo, já que todos os eventos estão acessíveis em qualquer altura e se podem obter através de uma localização conhecida, mas aumenta a carga do servidor em questão.

Relativamente à sincronização dos vector-versão, esta é feita periodicamente e consiste em enviar para um nó aleatório da rede de disseminação o vector-versão. Ao receber um vector-versão, as entradas são comparadas e verificam-se que eventos não são conhecidos. Os pedidos dos eventos em falta seguem o método apresentado anteriormente.

Mais uma vez, se existir um repositório remoto que armazene as notificações, a sincronização pode ser feita analisando o conteúdo do repositório, não sendo necessário a troca de mensagens entre os participantes do grupo, conseguindo-se assim interligar diferentes redes privadas sem a necessidade da existência de um nó público a fazer a ligação entre elas.

É importante sublinhar que a utilização das propriedades dos eventos evita que se transfira informação desnecessária. As respostas aos pedidos de eventos têm em conta quais os eventos realmente importantes, com base no último evento importante, conseguindo-se dessa forma minimizar as mensagens enviadas.

O sistema de repositórios e de *cache* fornece ainda suporte para a desconexão, ainda que limitado. Um nó tem a possibilidade de receber notificações que não processou durante o período de desconexão, mesmo que o emissor desses eventos não se encontre activo no grupo, desde que algum participante os mantenha na sua *cache*.

4.3.7 Outros aspectos importantes

Uma das funcionalidades disponibilizadas pelo sistema de disseminação é a possibilidade de enviar mensagens utilizando o caminho inverso, seguindo uma política *best-effort*. Durante a propagação de DisseminationEvent é guardado o caminho seguido pela mensagem. Quando a notificação chega a um nó, este pode enviar uma mensagem pelo caminho inverso, usando essa informação.

Se o nó que gerou a notificação for público, a comunicação é feita directamente. Se for privado encontra-se o primeiro nó público (que terá ligação com a rede local onde o evento foi gerado) e este reencaminha a mensagem para a rede local, que será encaminhada pelos nós da rede privada até chegar ao seu destino, no caso deste ainda se encontrar activo.

5 . Avaliação

O desenvolvimento do protótipo apresentado na secção anterior permite que se avalie experimentalmente as funcionalidades do Forby. Neste capítulo apresenta-se essa avaliação, sendo feita uma análise em termos qualitativos e quantitativos. Na análise qualitativa, pretende-se avaliar se a solução desenvolvida alcança os objectivos propostos. Na avaliação quantitativa, pretende-se determinar a eficiência dos módulos de detecção e disseminação.

5.1 Avaliação Qualitativa

Nesta secção é feita uma avaliação qualitativa das aplicações utilizadas para testar o protótipo implementado, procurando-se analisar a sua utilidade para as aplicações em causa. Esta análise procura demonstrar de que forma o Forby contribui para o desenvolvimento mais rápido e simples das aplicações que necessitam de utilizar ficheiros partilhados.

5.1.1 Aplicações

Para fazer a avaliação qualitativa do sistema foram desenvolvidas duas aplicações distintas, o P2PVC^2 e uma aplicação de replicação. Para ambas procura-se explicar de que forma o Forby foi usado para suportar o seu desenvolvimento.

5.1.1.1 P2PVC^2

O VC^2 [17] é um sistema de *awareness* que utiliza um servidor de CVS/SVN para disseminar informação relativa às versões dos ficheiros de trabalho de um grupo. Na sua implementação base, é feita uma monitorização da área de trabalho e são recolhidos eventos, resultantes de acessos ao sistema de ficheiros, que são enviados para o servidor remoto.

Cada participante, para ter conhecimento dos eventos gerados pelos restantes elementos do grupo, tem que fazer *polling* ao servidor, tornando a disseminação de eventos bastante ineficiente e impondo uma elevada carga no servidor de CVS/SVN. Esta situação levanta alguns problemas, principalmente quando se tratam de servidores públicos, como por exemplo, o *sourceforge*.

Neste trabalho criou-se uma nova versão deste sistema - o P2PVC^2 - recorrendo ao sistema Forby para fazer a disseminação da informação de *awareness* para que o sistema não imponha uma carga elevada no servidor central e possa ser utilizado numa escala maior.

As alterações feitas à versão base foram grandes, mas simples de efectuar, uma vez que o Forby fornece todas as funcionalidades necessárias ao funcionamento desta aplicação.

Para utilizar o Forby, definiram-se, em primeiro lugar, os parâmetros relativos ao sistema de detecção. A escolha da implementação do módulo EventReceiver é feita de acordo com o sistema de operação utilizado. No caso do protótipo implementado, as duas soluções - *minifilter* em conjunto com o TCPReceiver e JNotify - podem ser utilizadas.

De seguida, é necessário definir o EventFilter a utilizar. O filtro do P2PVC^2 mantém uma lista de ficheiros a monitorizar e para cada um desses ficheiros, o tipo de eventos que são importantes processar. Com esta informação, a validação de um FSEvent recolhido é simples, bastando verificar se o nome do ficheiro e o seu tipo correspondem ao esperado.

Se o evento for válido, é feito um processamento do mesmo, para ser ou não entregue ao EventManager e posteriormente encaminhado para a aplicação. Neste caso, são entregues os eventos do tipo FILE_WRITE e FILE_OPEN e FILE_CLOSE.

Os eventos de *create* e *delete* adicionam ou removem ficheiros do filtro. Os eventos de *open* e *close*, são usados para fazer optimizações ao nível do filtro. Se, por exemplo, houver uma sequência de eventos do tipo *open - write - write - close*, é apenas necessário enviar a última modificação para que os outros participantes saibam que houve uma alteração no ficheiro.

Relativamente aos eventos entregues à aplicação, os eventos do tipo FILE_WRITE são disseminados pelos restantes utilizadores. Os eventos de FILE_OPEN e FILE_CLOSE não são encaminhados para os restantes nós. Estes permitem ao P2PVC^2 determinar os ficheiros que estão a ser utilizados, podendo assim apresentar as notificações importantes para o utilizador a cada momento.

Uma vez que o P2PVC^2 trabalha sobre versões de ficheiros, é importante indicar se a modificação de um ficheiro corresponde a um *checkout* ou a uma alteração na versão local. Para isto o evento propagado para os outros nós - FSEvent - foi estendido para conter essa informação.

Quanto às propriedades de *commute* e *overwrite*, dado que o P2PVC^2 apenas dissemina eventos do tipo FILE_WRITE, só este tipo de eventos é armazenado na base de dados e assim só é necessário definir as propriedades para estes eventos. As propriedades definidas seguem o apresentado na Figura 5.1.

Commute:
if different filename

Overwrite:
if equal filename and event type equals FILE_WRITE

Figura 5.1 Métodos de *commute* e *overwrite* definidos para o P2PVC2.

Para dois eventos comutarem, basta que se refiram a ficheiros diferentes, dado que as escritas são independentes. Quando à propriedade de *overwrite*, um evento pode suprimir outros que reflictam alterações sobre o mesmo ficheiro. Como se pretende apenas disseminar a informação sobre a alteração dos ficheiros, para um nó saber que um ficheiro foi alterado necessita apenas de aceder ao último evento que indique essa alteração.

Por fim, a aplicação deve registar-se no EventManager para receber os eventos que lhe são destinados. Sempre que um evento é recebido do de detecção, este é encaminhado para o sistema de disseminação, para ser difundido pelos restantes elementos do grupo.

Na Figura 5.2 pode ver-se o método de inicialização do sistema de detecção.

```
private void initEventManager() {
  // Create the EventManager.
  IEventManager eventManager = new EventManager();

  // Initiates an EventReceiver
  new JNotifyEventReceiverWIN(path, eventManager);

  // Creates an EventFilter and adds the files to watch.
  IEventFilter eventFilter = new VC2Filter();
  addFilesToFilter(eventFilter, path);

  // Subscribes this application with the EventManager to receive the generated events.
  eventManager.subscribe(eventFilter, this);
}
```

Figura 5.2 Inicialização do sistema de detecção.

No que diz respeito ao sistema de disseminação, todas as opções são definidas no ficheiro de propriedades, onde se indicam todos os parâmetros necessários à criação da rede de disseminação.

É, no entanto, necessário definir o que fazer quando se recebe um DisseminationEvent. Com base nos ficheiros abertos a cada momento, as notificações que são recebidas são apresentadas, surgindo no ecrã uma janela com essa informação. No caso de se receberem notificações

sobre ficheiros que não estão a ser utilizados, estas são armazenadas para posteriormente se informar o utilizador da modificação dos ficheiros, quando estes forem acedidos.

Para suportar os pedidos de *commit* de ficheiros que estão a ser alterados por outros utilizadores, o P2PVC^2 utiliza o mecanismo de envio de mensagens pelo caminho inverso fornecido pelo Forby. O utilizador que está a modificar o ficheiro, ao receber essa mensagem, escolhe se quer ou não fazer *commit* da sua versão e envia uma mensagem para o emissor do pedido, indicando a sua escolha.

Como se pode verificar, a maior parte do esforço necessário à implementação do sistema P2PVC^2 está presente na inicialização do sistema de detecção, mais precisamente na criação de um filtro adequado à aplicação.

A utilização do Forby permitiu simplificar a criação do sistema e torná-lo mais portável, uma vez que as primitivas de recolha e disseminação de eventos já estão definidas, e podem ser utilizadas pelo P2PVC^2. A troca de mensagens é feita sem se recorrer ao servidor de controlo de versões e o número de notificações disseminadas é menor (ver secção 5.2.2), dadas as características do sistema de disseminação, detalhadas no capítulo 4.

5.1.1.2 Aplicação de replicação

A segunda aplicação de teste consiste num sistema de replicação simples, onde se pretende que o conteúdo de uma directoria seja replicado pelos vários elementos de um determinado grupo e que as alterações feitas sobre essa directoria sejam processadas em todos nós, sem ser efectuada qualquer validação das versões dos ficheiros. Esta aplicação pode ser usada, por exemplo, para partilhar fotografias entre vários utilizadores.

Tendo em conta as características requeridas por esta aplicação, é possível utilizar o Forby para fazer a captura e disseminação dos eventos produzidos em cada nó. A integração do Forby nesta aplicação é muito semelhante ao descrito para o P2PVC^2.

Para o sistema de detecção, é necessário escolher/definir o módulo de monitorização e captura dos eventos do sistema de ficheiros e criar um filtro específico para a aplicação. Tal como na aplicação anterior, o filtro tem a informação sobre os ficheiros e o tipo de eventos que são importantes para a aplicação. Neste caso, a monitorização é feita sobre o conteúdo de uma directoria definida pela aplicação.

Depois de validados e processados, a aplicação de replicação deve encaminhar para os restantes nós os eventos do tipo FILE_CREATE, FILE_DELETE e FILE_WRITE. Neste caso

não é necessário criar um tipo de evento específico para enviar, uma vez que a informação presente no FSEvent é suficiente para os requisitos desta aplicação.

As operações de comutação e de supressão de eventos, para esta aplicação, devem ter em conta os tipos de eventos disseminados, referidos anteriormente. Assim, os métodos seguem a implementação descrita nas Figuras 5.3 e 5.4.

Tipo: FILE_WRITE

Commute:
if different filename

Overwrite:
if equal filename and event type equals FILE_CREATE or FILE_WRITE

Figura 5.3 Métodos de *commute* e *overwrite* para eventos to tipo FILE_WRITE

Tipo: FILE_DELETE

Commute:
if different filename

Overwrite:
if equal filename and event type equals FILE_CREATE or FILE_WRITE

Figura 5.4 Métodos de *commute* e *overwrite* para eventos to tipo FILE_DELETE

A implementação das propriedades para esta aplicação segue a explicação apresentada na secção anterior, para o sistema P2PVC^2. Os eventos comutam se corresponderem a ficheiros diferentes. A supressão de eventos acontece quando os eventos se referem ficheiros idênticos e as operações efectuadas permitem a cada nó executar o processamento dos eventos para atingir um estado final consistente, sem ser necessário receber todos os eventos gerados. Por exemplo, um *delete* sobre o mesmo ficheiro suprime os eventos de *create* e *write* dado que, para apagar o ficheiro, não é necessário obter o seu conteúdo mais actual.

Quando um evento é recebido noutro nó do grupo, verifica-se o tipo de evento. Se for do tipo FILE_CREATE ou FILE_DELETE, esse nó pode criar ou remover o ficheiro em questão. Se for recebido um FILE_WRITE, então o nó tem que pedir o conteúdo do ficheiro ao emissor do evento.

Se o emissor for um nó público, estabelece-se uma ligação com esse nó e faz-se a transferência do conteúdo do ficheiro. Se o nó for privado, utiliza-se o envio de mensagens utilizando o caminho inverso, funcionalidade oferecida pelo sistema de disseminação. O nó privado, ao receber o pedido, vai abrir uma ligação para o nó que enviou a mensagem e transfere o ficheiro.

Se o nó que pede o ficheiro estiver numa rede privada, pode usar-se novamente o caminho inverso para propagar essa informação.

Nesta aplicação, escritas as concorrentes não são tratadas, mas seria possível integrar soluções de controlo de versões, adicionando a informação necessária a este controlo aos eventos propagados, como relógios vectoriais.

Mais uma vez, a utilização do Forby para recolher e disseminar a informação recolhida em cada nó simplifica a implementação desta aplicação. Permite também que participantes que utilizem diferentes sistemas de ficheiros/operação se liguem ao mesmo grupo, sem ser necessário produzir código específico para cada sistema.

5.2 Avaliação Quantitativa

A avaliação quantitativa pretende medir o desempenho das componentes do sistema, para avaliar a eficiência do mesmo. Esta análise será feita em termos do *overhead* introduzido pelas várias formas de monitorização do sistema de ficheiros e pelo número de mensagens geradas, no sistema de disseminação.

5.2.1 Sistema de detecção

Nesta secção é analisado o *overhead* introduzido pelas diferentes implementações do módulo Event Receiver, que faz a recolha dos eventos gerados por acessos ao sistema de ficheiros.

Para se determinar os valores do overhead, efectuaram-se as seguintes operações: descompressão de um ficheiro de extensão RAR com 149 MB, seguindo-se a compressão dos ficheiros extraídos, finalizando-se com a remoção de todos os ficheiros utilizados. O arquivo é constituído por 969 ficheiros, 137 directorias. Os ficheiros têm dimensão reduzida, variando entre os 59 bytes e 2 MB, situando-se a maioria dos ficheiros na casa nos KB.

A sequência de operações indicada foi executada trinta vezes num sistema com processador Intel Pentium M a 1,86GHz, com 1,00 GB de memória RAM. Na Tabela 5.1 apresentam-se os

tempos médios de execução obtidos para os vários tipos de monitorização.

Tipo de monitorização	Duração média das execuções (em segundos)	*Overhead* da solução (em segundos)	*Overhead* da solução (em percentagem)
Sem monitorização (Windows)	128	-	-
Jnotify (Windows)	134	6	4,7
Minifilter (API Installable file systems)	196	68	53,1
Sem monitorização (Linux)	120	-	-
Jnotify (Linux)	138	18	15

Tabela 5.1 Tempos de execução e *overhead* para os diferentes tipos de monitorização.

Como se pode ver na Tabela 5.1, o *overhead* imposto pela utilização do Jnotify como sistema de monitorização não é muito significativo, acrescentando 6 segundos ao tempo de execução médio sem monitorização, o que corresponde a um aumento de cerca de 4,7% do tempo de execução.

Por sua vez, a utilização do driver da API Installable Filesystems incrementa, em média, 68 segundos ao tempo de execução, o que representa um aumento de cerca 53%, tornando o overhead desta solução bastante elevado.

Os resultados em ambiente Linux servem para comparar o *overhead* introduzido pela mesma solução (Jnotify) em ambientes distintos. Como se pode observar, ao utilizar este sistema de monitorização, o tempo médio de execução aumenta 18 segundos, o que corresponde a um aumento de 15%.

Com os resultados apresentados pode então concluir-se que a utilização do Jnotify, em ambiente Windows consegue bons resultados comparando com o minifilter. Quanto utilizado em ambientes diferentes, o mesmo sistema de monitorização apresenta resultados distintos, sendo mais eficiente a sua utilização em Windows. Esta diferença deve-se à eficiência da API Win32, conforme indicado pelo autor do sistema Jnotify [1].

Com base nestes resultados, pode concluir-se que o overhead introduzido pelo sistema de detecção é reduzido, com excepção do *minifilter*. Para avaliar se este *overhead* seria aceitável

num ambiente interactivo, é necessário proceder-se a novas experiências.

Assim, foi também determinado o *overhead* do acesso a um ficheiro, para determinar qual o impacto da monitorização em ambiente interactivo. Para tal, executaram-se 500 leituras e escritas em ficheiros de diferentes dimensões. Nas tabelas 5.2 e 5.3 apresentam-se os tempos de execução médios para cada leitura/escrita de um ficheiro completo.

	Tempo de execução (ms)		
Tamanho do ficheiro	Sem monitorização	*JNotify*	*minifilter*
500 KB	17	18	17
1 MB	38	38	38
10 MB	485	485	491
20 MB	966	990	977

Tabela 5.2 Tempos de execução médio para uma escrita.

	Tempo de execução (ms)		
Tamanho do ficheiro	Sem monitorização	*JNotify*	*minifilter*
500 KB	1	1	1
1 MB	3	3	3
10 MB	39	39	40
20 MB	80	81	84

Tabela 5.3 Tempos de execução médio para uma leitura.

As operações de escrita são influenciadas pela gestão que o sistema de operação faz do sistema de ficheiros, o número de processos a utilizar o disco ou os blocos onde os dados são escritos. Isto faz com que não se consiga determinar com rigor o overhead imposto em cada experiência.

No entanto, com base nos resultados apresentados pode concluir-se que o overhead, imposto

no acesso aos ficheiros ao utilizar um dos mecanismos de monitorização implementados, não é significativo e é desprezável num ambiente interactivo.

5.2.2 Sistema de disseminação

A avaliação quantitativa do sistema de disseminação baseia-se na análise do número de mensagens enviadas pelo sistema, para propagar as mensagens para os participantes de um grupo. Para isso, será feita uma comparação da implementação VC^2 com a versão que utiliza o Forby (P2PVC^2).

Para esta avaliação não foi medido o tempo de disseminação, porque este depende de diversos factores, como as condições e a sobrecarga da rede, o número de participantes, a latência entre os nós do grupo, entre outros.

É importante referir que nesta análise se omitem as comunicações necessárias à propagação da informação relativa ao estado do grupo, como as entradas e saídas, uma vez que estes dados são variáveis.

Cenários de aplicação

Para avaliar o sistema, foram definidos três cenários, que influenciam o número de mensagens enviadas para o servidor e trocadas entre os clientes. Estes cenários são:

1. **Utilização versão original do VC^2** - Conforme apresentado na secção 2.2.1, o sistema VC^2 recorre ao servidor de CVS/SVN para fazer a disseminação da informação de *awareness* recolhida em cada área de trabalho. A sua implementação passa por fazer *polling* periódico ao servidor para enviar as notificações e verificar se existem novos eventos ou pedidos para serem processados.

2. **Utilização do P2PVC^2, com servidor central (P2PVC^2-SC)** - Esta implementação utiliza o servidor como ponto de encontro, permitindo aos nós entrarem na rede de disseminação. As notificações são enviadas pela árvore de disseminação sem necessidade de contactar o servidor. No entanto, o servidor também pode ser utilizado como repositório dos eventos gerados.

3. **Utilização do P2PVC^2, com DHT (P2PVC^2-DHT)** - Nesta implementação não existe um servidor remoto. A entrada no grupo de disseminação é feita recorrendo-se a um

rendez-vous point acessível pelo envio de mensagens utilizando uma DHT.

Análise das mensagens periódicas enviadas

Com base nos cenários apresentados anteriormente, começa-se por apresentar as comunicações periódicas trocadas entre os elementos do sistema (incluindo o servidor).

No VC^2, para cada ficheiro modificado localmente e enquanto não é feito o *commit*, o VC^2 vai verificar periodicamente o ficheiro de meta-informação associado a esse ficheiro, para determinar se outro cliente pediu que a versão actual fosse submetida ao servidor. Esta verificação implica uma comunicação periódica com o servidor.

Assumindo que se verifica o estado do ficheiro a cada 15 segundos (valor por omissão), o número de comunicações com o servidor é 4 por minuto, para cada elemento do grupo de trabalho que altere localmente algum ficheiro. Assim, o servidor vai ser contactado 4 * *n* vezes por minuto, com *n* a representar o número de utilizadores do sistema com ficheiros modificados.

Para reduzir o número de comunicações, seria possível aumentar o intervalo de verificação dos ficheiros. No entanto, isso iria impedir a aplicação de atingir os seus objectivos, uma vez que o pedido de *commit* deixaria de chegar a cada utilizador em tempo útil.

No P2PVC^2-SC, não são trocadas mensagens periódicas com o servidor. No entanto, para que os participantes tolerem eventuais falhas e mantenham uma visão geral actualizada do conjunto dos eventos disseminados, os nós trocam os seus vectores-versão com um nó aleatório, como forma de garantir sincronização. Por omissão, este processo é executado a cada 10 minutos, pelo que o número de comunicações efectuadas por cada nó é 0,1/minuto.

Por fim, no P2PVC^2-DHT, para além da sincronização temporal dos relógios vectoriais, apresentada para o segundo cenário, existe a necessidade de contactar periodicamente o *rendez-vous* para confirmar que este se mantém activo. Esta comunicação é feita apenas pelo primeiro elemento do grupo, a cada 5 minutos, representando 0,2 comunicações por minuto. Assim, para este cenário, no pior caso, existem 0,3 comunicações/minuto.

Na tabela 5.4 resume-se o número comunicações periódicas em cada um dos cenários.

Com base nos resultados apresentados, pode concluir-se que o número de mensagens comunicações efectuadas pelo VC^2, comparando com os restantes, é bastante elevado. Ao transpor estes cenários para uma utilização real, com múltiplos utilizadores, pode afirmar-se que o primeiro vai sobrecarregar o servidor com o aumento do número de participantes, sendo a

Cenários	Número de comunicações / Minuto (servidor)	Número de comunicações / Minuto (participante)
VC^2	4 * n	4
P2PVC^2 - SC	-	0,1
P2PVC^2 - DHT	-	0,3

Tabela 5.4 Número de comunicações periódicas.

solução menos escalável.

Análise das mensagens enviadas por operação

Agora, estudam-se as comunicações efectuadas durante o acesso a um ficheiro. Para tal, vamos considerar um caso da escrita de um ficheiro que inclui uma sequência de operações de *write* (por exemplo, *open - write - write - write - close*).

No VC^2, é feita uma comunicação com o servidor, enviando o ficheiro de meta-informação modificado, para indicar que o ficheiro foi alterado.

No caso do P2PVC^2-SC, esta informação é disseminada pelos participantes sem se recorrer ao servidor, o que leva o emissor a fazer entre 0 e 3 comunicações, consoante o número de nós com quem se encontra ligado, determinados pelo algoritmo de criação da árvore de disseminação. No entanto, caso o servidor seja utilizado como repositório de notificações, o emissor deve estabelecer uma ligação para propagar o novo evento, o que corresponde, tal como no cenário anterior, a uma comunicação com o servidor.

Com o P2PVC^2-DHT apenas se envia a notificação para os restantes elementos do grupo.

Em todos os cenários, caso se pretenda pedir a versão mais actual, alterada por outro participante, estabelecem-se duas comunicações adicionais, uma para o pedido e outra para a resposta.

Relativamente às operações de escrita no P2PVC^2, é importante referir que, tal como explicado anteriormente, o módulo de detecção implementado para este protótipo, no sistema Windows, não detecta os eventos de *open* e *close*. Isto impede que se faça uma optimização directa sobre o número de eventos a enviar, ou seja, cada escrita sucessiva gera um evento. Para contornar esta situação, o filtro agrupa os *writes* efectuados sobre o mesmo ficheiro por período

de tempo, conseguindo-se assim um evento por operações de escrita seguidas.

No sistema Linux, não existe este problema, dado que é possível recolher os eventos necessários para filtrar as escritas efectuadas ente o open e o close do ficheiro em questão.

A Tabela 5.5 apresenta um resumo do número de comunicações por escrita.

Cenários	Número de mensagens (servidor)	Número de mensagens (participante)
VC^2	1	1
$P2PVC^2$ - SC	1	[0 – 3]
$P2PVC^2$ - DHT	-	[0 – 3]

Tabela 5.5 Número de comunicações numa operação de escrita.

Ao considerar uma operação de leitura (*open* - *read* - *close*), também existe diferença no número de comunicações com o servidor, consoante o cenário de aplicação. Com o VC^2, estas operações geram uma comunicação para verificar se o ficheiro que está a ser acedido é o mais actual.

Nos outros dois cenários, não é necessário enviar qualquer mensagem pela rede, uma vez que todas as notificações geradas são armazenadas em cada nó, tal como descrito na secção 5.1.1.1. Assim, a aplicação tem apenas que verificar se para o ficheiro lido foi recebida alguma notificação de alteração.

A Tabela 5.6 apresenta os resultados relativos a uma operação de leitura.

Cenários	Número de mensagens (servidor)	Número de mensagens (participante)
VC^2	1	1
$P2PVC^2$ - SC	0	0
$P2PVC^2$ - DHT	-	0

Tabela 5.6 Número de comunicações numa operação de escrita.

Conclusão

Com base nos resultados apresentados nesta secção, e em forma de comentário final ao número de comunicações estabelecidas em cada um dos cenários, pode verificar-se que, no que respeita às mensagens por operação de leitura e escrita, não existe uma diferença significativa em relação a cada um dos cenários de utilização.

No entanto, ao considerar as mensagens periódicas, a utilização do VC^2, implica realizar um grande número de comunicações com o servidor central, situação que pode levar à sobrecarga desse servidor à medida que o número de participantes aumenta.

6 . Conclusões e trabalho futuro

Neste capítulo apresentam-se as conclusões finais acerca do trabalho realizado e indicam-se direcções para o trabalho futuro.

6.1 Conclusões

A partilha de ficheiros entre grupos de utilizadores é uma situação comum, com diversos cenários de utilização possíveis. Estes levam a diferentes requisitos do sistema de suporta à partilha de informação. Uma funcionalidade base para criar este suporte diz respeito à recolha, tratamento e disseminação de informação sobre acções na área de trabalho de cada participante do grupo.

Dado o potencial número de diferentes aplicações para suportar os vários tipos de cenários de utilização, é interessante fornecer mecanismos genéricos que possam ser integrados nessas aplicações, simplificando o seu desenvolvimento e melhorando a sua eficiência.

Assim, o trabalho realizado nesta dissertação procurou resolver alguns dos problemas associados à monitorização dos ficheiros e disseminação das respectivas notificações. Com esse objectivo foi implementado o Forby, um sistema genérico e simples de utilizar que oferece uma interface de captura e disseminação de eventos relativos ao acesso a ficheiros.

Podem considerar-se como as características mais importantes deste sistema, no módulo de detecção, a possibilidade de utilizar diversos mecanismos de monitorização da área de trabalho e integrar futuramente outros mecanismos que melhorem a eficiência da recolha da informação. A API deste módulo é simples de utilizar e integrar em qualquer aplicação.

A criação de filtros, seguindo uma interface específica, para processar a informação recolhida reduz a complexidade da aplicação final e possibilita optimizar a informação que chega à aplicação.

No que diz respeito ao módulo de disseminação, algumas das suas principais características distinguem o Forby dos restantes sistemas genéricos de disseminação de eventos. Algumas dessas características são:

– A sua capacidade de minimizar as mensagens enviadas, tirando partido da semântica dos eventos disseminados de forma a minimizar o número de mensagens propagadas;

- O suporte integrado para utilizadores ligados a redes privadas, sem necessidade de modificar as políticas de gestão da rede para trocar informação;

- Utilização de vários tipos de ponto de encontro que se adaptam aos recursos disponibilizados pela aplicação, fornecendo um suporte para a criação de grupos de disseminação mesmo que não existam servidores de apoio;

- A possibilidade de utilizar servidores como o SVN ou o *Gmail* como repositórios de dados para armazenar as notificações disseminadas, fornecendo-se assim um mecanismo simples para suporte à desconexão e para permitir a comunicação entre redes privadas.

As aplicações desenvolvidas para demonstrar a funcionalidade do Forby permitem concluir que o sistema permite lidar de forma simples com os principais problemas enfrentados pelas aplicações que utilizam ficheiros partilhados.

Adicionalmente, este sistema facilita bastante o desenvolvimento desta classe de aplicações. Os resultados de desempenho obtidos permitem afirmar que *overhead* imposto pelo sistema de monitorização é aceitável e no caso específico do VC^2, foi possível melhorar o seu problema de escalabilidade, minimizando o número de mensagens que passam pelo servidor.

6.2 Trabalho Futuro

Como trabalho futuro, propõem-se algumas melhorias e optimizações que se podem efectuar no protótipo do Forby, com vista a criar um sistema mais robusto, que possa ser utilizado num ambiente real. De seguida apresentam-se essas propostas de evolução do Forby.

No que diz respeito ao sistema de monitorização, a implementação de outros mecanismos de recolha de eventos, para os diversos sistemas de operação, seria importante para se tentar chegar a uma solução mais eficiente, apesar dos resultados satisfatórios obtidos com o JNotify. Em particular, é interessante explorar os *minifilters*, procurando melhorar o suporte oferecido em Windows, tentando-se obter mais informação sobre os acessos aos ficheiros.

Também seria importante fornecer uma interface para modificar os resultados dos acessos ao sistema de ficheiros, por exemplo, para desviar uma chamada ao sistema e processá-la numa aplicação ao nível de utilizador. Isto permitia, por exemplo, que uma aplicação pudesse fazer a sincronização dos ficheiros no momento anterior à sua abertura.

Neste trabalho, com a utilização dos *minifilters*, tentou explorar-se essa situação, mas chegou-se à conclusão que seria demasiado complexo e daí ser apresentado como trabalho futuro.

No que diz respeito à disseminação, uma das principais melhorias a efectuar seria a introdução de mecanismos de segurança que permitissem garantir a privacidade, autenticidade e validade das mensagens trocadas pelos participantes. Também o controlo de acesso ao grupo de disseminação seria uma mais-valia.

Relativamente à rede de disseminação, seria importante analisar e implementar algoritmos optimizados que permitam criar a rede de utilizadores e melhorar a eficiência do encaminhamento das mensagens nessa rede.

Bibliografia

[1] Jnotify, 2005. http://jnotify.sourceforge.net/.

[2] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in p2p systems. *Commun. ACM*, 46(2):43–48, 2003.

[3] Bazaar. Bazaar version control, 2007. http://bazaar-vcs.org/.

[4] Per Cederqvist et al. Version management with cvs, October 2007. http://ximbiot.com/cvs/cvshome/docs/.

[5] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, New York, NY, USA, 1987. ACM.

[6] Prasun Dewan and Rajesh Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proc. of the European Conference on Computer-Supported Cooperative Work - ECSCW'07*, pages 159–178, 2007.

[7] Dokan. Dokan library, 2009.

[8] Paul Dourish and Victoria Bellotti. Awareness and coordination in shared workspaces. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 107–114, New York, NY, USA, 1992. ACM.

[9] Elvin. elvin.org, 2007. http://elvin.org/.

[10] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[11] Geraldine Fitzpatrick, Paul Marshall, and Anthony Phillips. Cvs integration with notification and chat: lightweight software team collaboration. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 49–58, New York, NY, USA, 2006. ACM.

[12] Susanne Hupfer, Li-Te Cheng, Steven Ross, and John Patterson. Introducing collaboration into an application development environment. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 21–24, New York, NY, USA, 2004. ACM.

[13] Claudia-Lavinia Ignat, Stavroula Papadopoulou, Gérald Oster, and Moira C. Norrie. Providing awareness in multi-synchronous collaboration without compromising privacy. In *CSCW '08: Proceedings of the ACM 2008 conference on Computer supported cooperative work*, pages 659–668, New York, NY, USA, 2008. ACM.

[14] Peter Jalbert. Google docs and spreadsheets, January 2008. `http://www.googletutor.com/2008/01/23/real-time-collaboration-with-google-docs-and-spreadsheets/`.

[15] Anthony Lamarca, W. Keith Edwards, Paul Dourish, John Lamping, Ian Smith, and Jim Thornton. Taking the work out of workflow: Mechanisms for document-centered collaboration. In *Proceedings of the European Conf. Computer-Supported Cooperative Work ECSCW'99*, pages 1–20. Kluwer, 1999.

[16] J. Legatheaux Martins and S. Duarte. Routing Algorithms for Content-based Publish/Subscribe Systems. *IEEE Communications Tutorials and Surveys (Accepted for Publication)*, page 21, 2009.

[17] D. Machado, N. Preguiça, C. Baquero, and J. Legatheaux Martins. Vc2 - providing awareness in off-the-shelf versioncontrol systems. In *IWCES9: Proc. of the Nineth International Workshop on Collaborative Editing Systems*. IEEE Computer Society, 11 2007.

[18] Microsoft. Installable file system, 2009. `http://msdn.microsoft.com/enus/library/dd446412.aspx`.

[19] Pascal Molli, Hala Skaf-molli, Christophe Bouthier, and Loria Inria Lorraine. State treemap: an awareness widget for multi-synchronous groupware. In *International Workshop on Groupware*, pages 106–114, 2001.

[20] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, 1983.

[21] Daniel Peek and Jason Flinn. Ensemblue: integrating distributed storage and consumer electronics. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 219–232, Berkeley, CA, USA, 2006. USENIX Association.

[22] B. C. Pierce. Unison file synchronizer, 2009. `http://www.cis.upenn.edu/~bcpierce/unison/index.html`.

[23] Peter R. Pietzuch and Jean Bacon. Hermes: A distributed event-based middleware architecture. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[25] Anita Sarma, Zahra Noroozi, and André van der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Peter Sutton, Rhys Arkins, and Bill Segall. Supporting disconnectedness-transparent information delivery for mobile and invisible computing. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 277, Washington, DC, USA, 2001. IEEE Computer Society.

[27] SVK. The svk version control system, 2006. `http://svk.bestpractical.com/view/HomePage`.

[28] svn. Subversion. next-generation open source version control, March 2009. `http://subversion.tigris.org/`.

[29] Walter F. Tichy. Rcs—a system for version control. *Softw. Pract. Exper.*, 15(7):637–654, 1985.

[30] Kaushik Veeraraghavan, Edmund B. Nightingale, Jason Flinn, and Brian Noble. qufiles: a unifying abstraction for mobile data management. In *HotMobile '08: Proceedings of the*

9th workshop on Mobile computing systems and applications, pages 65–68, New York, NY, USA, 2008. ACM.

[31] Wikipedia. Filesystem in userspace — wikipedia, the free encyclopedia, 2007. `http://en.wikipedia.org/w/index.php?title=Filesystem_in_Userspace\&oldid=133824298`.

[32] Wikipedia. Inotify — wikipedia, the free encyclopedia, 2007. `http://en.wikipedia.org/w/index.php?title=Inotify&oldid=135783317`.

[33] D.J. Worth and C. Greenough. Comparison of cvs and subversion, October 2005. [Online; accessed October-2005].

[34] Erez Zadok and Jason Nieh. Fist: a language for stackable file systems. In *Proc. of the Annual Technical Conference on 2000 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2000. USENIX Association.