Sistemas de Arquivos Distribuídos

Capítulo 8

8.1 Introdução

8.2 Arquitetura do Serviço de Arquivos

8.3 Sun Network File System (NFS)

8.4 Avanços recentes

8.5 Sumário

Objetivos

- Entender os requisitos que afetem o projeto de um SA distribuído
- NFS: entender como é projetado
  - Caching é uma técnica de projeto essencial
  - Segurança requer atenção especial
- Avanços recentes

*

Sistemas de armazenamento e suas propriedades

- Na primeira geração de SAD (1974-95), os SAs (ex. NFS) eram apenas sistema de armazenamento em rede
- Com o advento de sistemas distribuídos orientados a objetos e a Web, começou a ficar mais complexo.

*

Storage systems and their properties

Figure 8.1

Types of consistency between copies: 1 - strict one-copy consistency
√ - approximate consistency
X - no automatic consistency

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Memória principal	×	×	×	1	RAM
sistema de ficheiros	×	✓	×	1	UNIX file system
Sistema de arquivos distribuído	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Memória compartilhada distribuída	✓	×	✓	✓	Ivy (Ch. 16)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Persistent distributed object store	✓	✓	✓	✓	PerDiS, Khazana

*

O que é um sistema de arquivos ? *1*

- Conjuntos de dados armazenados de forma persistente
- Espaço de nomes hierárquico visível a todos os processos
- API com as seguintes características:
  - Operaçoes de acesso e atualização nos conjuntos de dados persistentes
  - Modelo de acesso sequencial (com alguns recursos de acesso desordenado)
- Compartilhamento de dados entre usuarios com controle de acesso
- Acesso concorrente:
  - Funciona bem para acessos em modo de leitura
  - A modificação de dados pode levar a inconsistências
- Outras funcionalidades:
  - Depósitos de dados “montáveis”
  - O que mais ? ...

*

O que é um sistema de arquivos? 2

Figure 8.4 Operações com sistemas de arquivos Unix

filedes = open(name, mode) *filedes = creat(name, mode)*	Abre um arquivo existente dado um nome Cria um arquivo com um nome qualquer Ambas operações retuornam uma referência para o arquivo Aberto. mode é *read*, *write* ou ambos.
status = close(filedes)	Fecha o arquivo referenciado por *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfere n bytes do arquivo referenciado por *filedes* ao *buffer*. Transfere *n* bytes do buffer para para o arquivo refer. por *filedes* Ambas operações retornam o número de bytes transferidos E avançam o ponteiro de leitura/escrita..
pos = lseek(filedes, offset, whence)	Move o ponteiro de leitura/escrita para a posição offset
status = unlink(name)	R R
status = link(name1, name2)	C
status = stat(name, buffer)	C

Exercicio em Classe

Fazer um programa em C que abra um arquivo para escrita, escreva seu nome nele e feche o arquivo. Depois abra-o e exiba o seu conteúdo. Use as operações na tabela 8.4

*

O que é um sistema de arquivos?
(uma estrutura modular tipica para implementação de um SA)

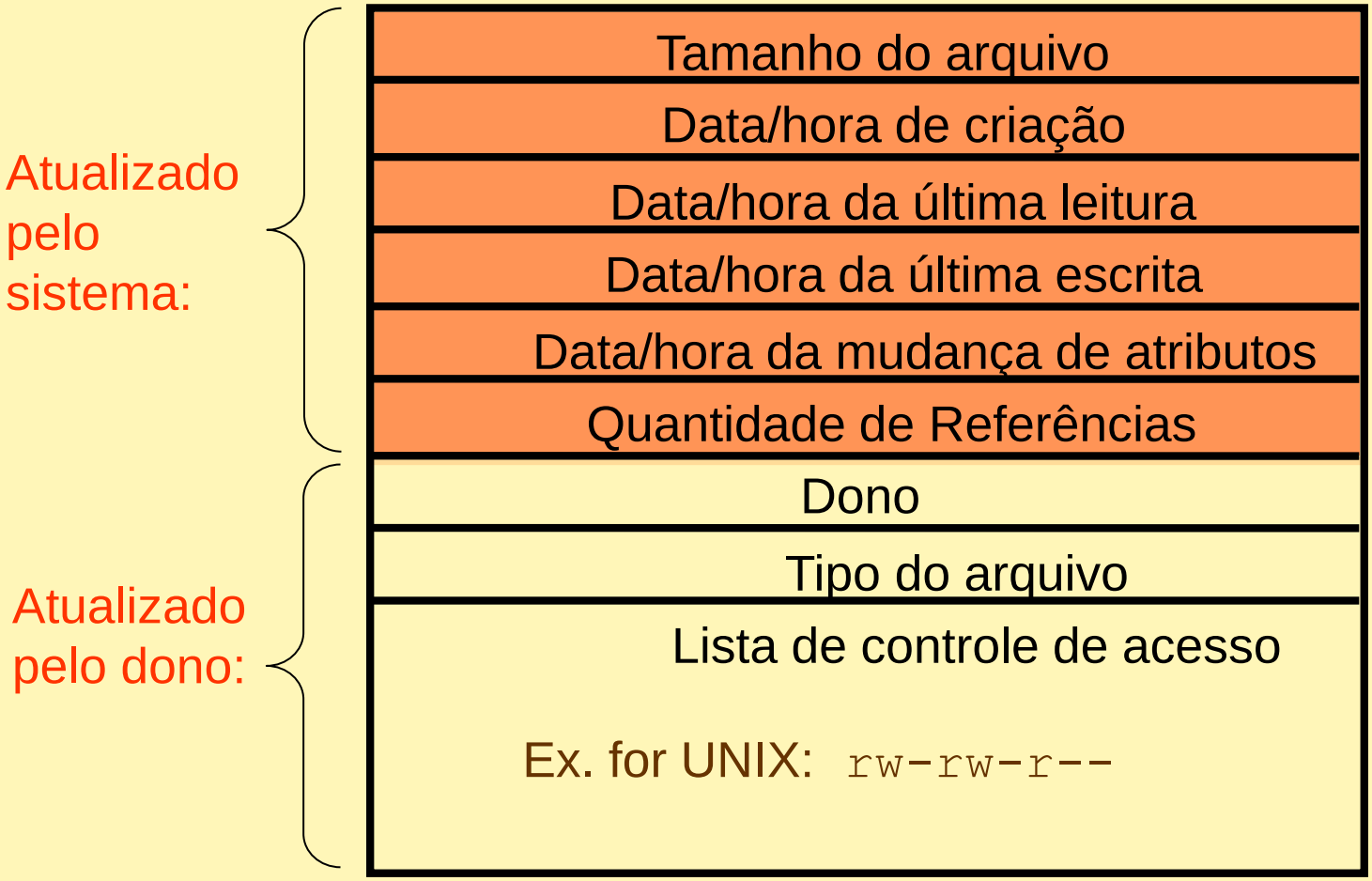
Figure 8.2 Modulos do SA (não-distribuído)

de Diretório:	Relaciona nome do arquivo com seu ID (FID)
de Arquivos:	Relaciona FIDs a arquivos em disco
de Controle de Acesso:	Checa permissões de operações solicitadas
de acesso a arquivos	Lê e escreve informações nos arquivos
de blocos	Acessa e aloca informações em blocos
De dispositivo	E/S com discos e bufferização

*

O que é um sistema de arquivos ? 4

Figure 8.3 Estrutura do registro com atributos do arquivo



*

Requisitos de um serviço/sistema de arquivos distribuídos

- Transparência
- Concorrencia
- Replicação
- Heterogeneidade
- Toler. a falhas
- Consistencia
- Segurança
- Eficiencia..

Tranparencias

Acesso: Mesmas operações (programas cliente não sabem sobre distrib. dos arquivos)

Localização: Mesma nomeação após relocação de arquivos ou processos (programa cliente deve ver nomeação uniforme)

Mobilidade: Relocação automatica de arquivos (nem programas cliente nem tabelas de administração precisam ser mudados quando arquivos são movidos).

Performance: Performance satisfatória em uma faixa específica de carga do sistema

Escalabilidade: Serviço pode expandir para atender crescimento necessário.

Sistema de arquivo o serviço Mais sobrecarregado em uma Intranet. Sua performance e funcionalidade são críticos

*

Requisitos de um serviço/sistema de arquivos distribuídos

- Transparência
- Concorrencia
- Replicação
- Heterogeneidade
- Toler. a falhas
- Consistencia
- Segurança
- Eficiencia..

Mudanças em um arquivo realizadas por um programa cliente não deve interferir com a operação de outros clientes acessando simultaneamente ou mudando o mesmo arquivo

Propriedades de concorrência

- Isolação

- Trava em nível de arquivo ou de registro

- Outras formas de controle de concorrência para minimizar contenção*

* Contenção: conflito ao acessar recurso compartilhado

Sistema de arquivo o serviço Mais sobrecarregado em uma Intranet. Sua performance e funcionalidade são críticos

*

Requisitos de um serviço/sistema de arquivos distribuídos

- Transparência
- Concorrencia
- Replicação
- Heterogeneidade
- Toler. a falhas
- Consistencia
- Segurança
- Eficiencia..

Serviço/sistema de arquivos mantem múltiplas cópias idênticas de arquivos

- Compartilham carga entre os servidores para tornar o serviço mais escalável
- Acesso local tem melhor tempo de resposta (lower latency)
- Tolerância a falhas

Replicação total é dificil de implementar.

Uso de caches (do arquivo todo ou de partes garante os maiores benefícios (exceto tolerância a falhas)

Sistema de arquivo o serviço Mais sobrecarregado em uma Intranet. Sua performance e funcionalidade são críticos

*

Requisitos de um serviço/sistema de arquivos distribuídos

- Transparência
- Concorrencia
- Replicação
- Heterogeneidade
- Toler. a falhas
- Consistencia
- Segurança
- Eficiencia..

Propriedades de Heterogeneidade

Serviço pode ser acessado por clientes executando qualquer (ou quase) SO ou plataforma de hardware.

O projeto deve ser compatível com sistemas de arquivos em diferentes SOs

Interfaces de serviço devem ser abertas. Especificações precisas de API foram publicadas.

Sistema de arquivo o serviço Mais sobrecarregado em uma Intranet. Sua performance e funcionalidade são críticos

*

Requisitos de um serviço/sistema de arquivos distribuídos

- Transparência
- Concorrencia
- Replicação
- Heterogeneidade
- Toler. a falhas
- Consistencia
- Segurança
- Eficiencia..

Sistema de arquivo o serviço Mais sobrecarregado em uma Intranet. Sua performance e funcionalidade são críticos

Consistência

Unix oferece semântica cópia-única para operações em arquivos locais.

Caching é completamente transparente.

Difícil de atingir a mesma semântica para SADs e oferecer boa performance e escalabilidade

*

Requisitos de um serviço/sistema de arquivos distribuídos

- Transparência
- Concorrencia
- Replicação
- Heterogeneidade
- Toler. a falhas
- Consistencia
- Segurança
- Eficiência

Sistema de arquivo o serviço Mais sobrecarregado em uma Intranet. Sua performance e funcionalidade são críticos

Segurança

Deve manter controle de acesso e privacidade como em arquivos locais.

- Baseado na identidade do usuário fazendo a solicitação
- Identidade de usuários remotos devem ser autenticadas
- Privacidade requer comunicação segura

Interfaces de serviço são abertas a todos os processos não excluídos de um firewall.

- Vulnerável a ataques de personificação (hacker invade um user ID) e outros

Requisitos de um serviço/sistema de arquivos distribuídos

- Transparência
- Concorrencia
- Replicação
- Heterogeneidade
- Toler. a falhas
- Consistencia
- Segurança
- Eficiencia..

Eficiencia

Objetivo de um SAD (SA Distribuído) é ter uma performance comparável com um SAL (SA Local).

Sistema de arquivo o serviço Mais sobrecarregado em uma Intranet. Sua performance e funcionalidade são críticos

*

Semânticas

- O que é Semântica nesse caso ?

  É o comportamento que devemos esperar do Sistema de Arquivos

- Quando há mais de um processo lendo/escrevendo em um arquivo
  - Controle de concorrência (ok!)
  - Semântica de compartilhamento
- Quatro tipos básicos
  - Semântica UNIX – cópia-única
  - Semântica de sessão
  - Arquivos imutáveis
  - Transações

Semântica UNIX – cópia-única

- Quando um READ ocorre após um WRITE, o READ retorna o valor que acabou de ser escrito
  - Obrigatoriedade de ordenação total
  - Sempre retorna o valor mais atual
- Implementação trivial
  - Servidor central que processa requisições na ordem (lógica)
    - *É necessário um servidor central*
- Pode gerar problemas graves de performance
  - Gera muita contingência, ou seja, o acesso deve ser sequencial
  - Processos aguardando muito para acessar o arquivo
    - *Serialização no acesso*

Semântica de sessão

- Servidor centralizado
  - traz problemas de gargalo e ponto central de falha
- Gargalo pode ser aliviado com uso de caches nos clientes
  - C1 lê o arquivo (coloca no cache) e altera
  - C2 lê o arquivo → dados antigos
- Solução → propagar alterações
  - Conceitualmente simples
  - Ineficiente

Semântica de sessão

- Relaxamento na semântica UNIX
- Mudanças em um arquivo são visíveis imediatamente para o processo, e para os demais apenas quando o arquivo for fechado
- Está errado?
  - Não
  - Amplamente implementado

Arquivos imutáveis

- Por que se preocupar?
  - Cada cliente teria acesso a sua cópia do arquivo
- Arquivos não podem ser alterados
  - Apenas READ e CREATE
- Alterar um arquivo?
  - Não pode, mas pode criar um novo de forma atômica
  - Arquivos são imutáveis, mas diretórios não

Arquivos imutáveis

- Dois processos desejando “alterar”
  - O mais novo altera o mais velho
  - Forma não determinística
- Se for alterar com alguém lendo?
  - Criar uma cópia do antigo para continuar lendo
  - Retornar erro para o leitor

Transações

- Tratar operações de forma atômica
  - Uma transação
- Dois processos ao mesmo tempo?
  - Transações permitem concorrência
  - Sistema/Servidor faz serialização
- Abordaremos em uma aula específica sobre isso

Semânticas

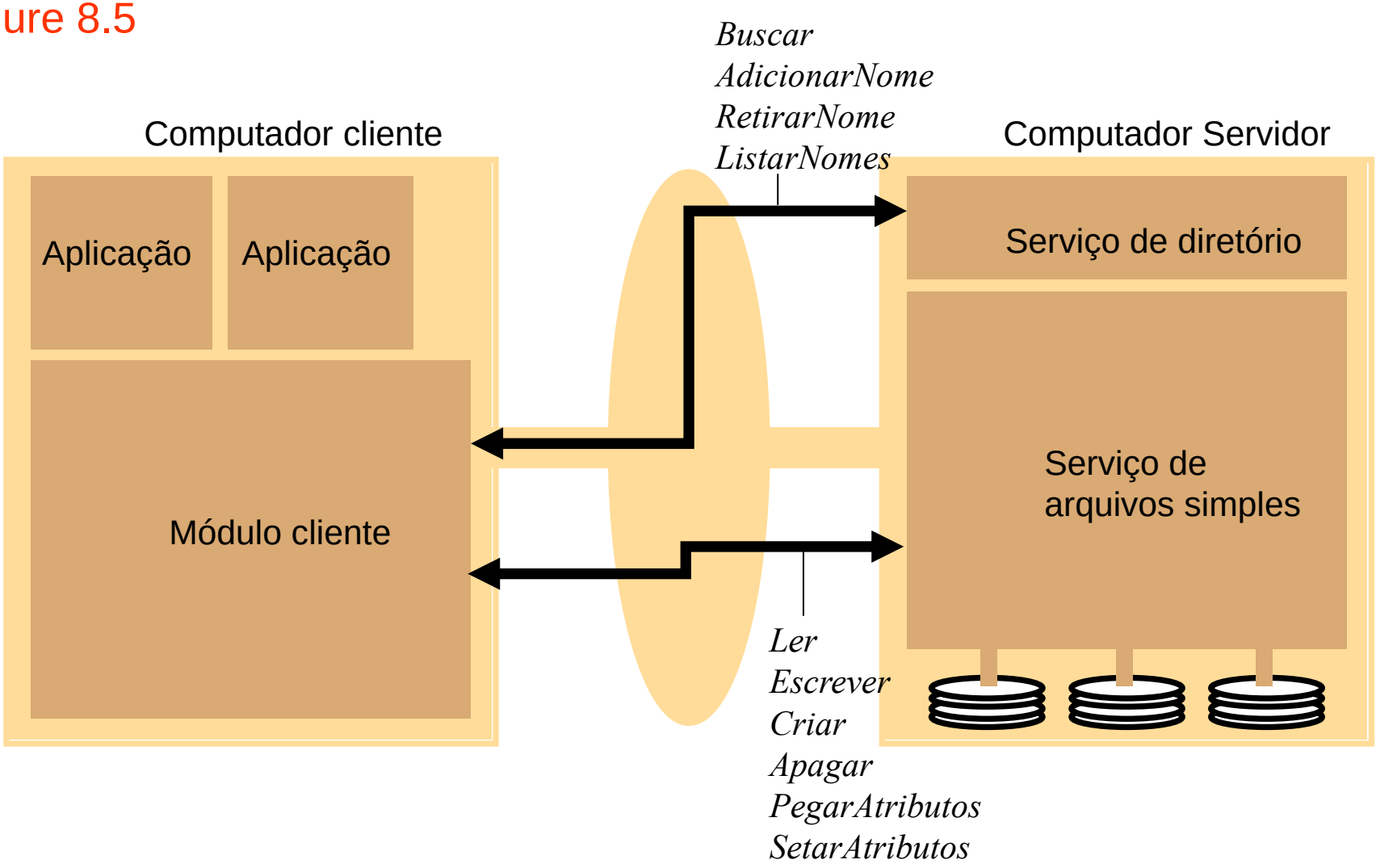
- São apenas diretrizes para construção
  - “Meu SA tem semântica Unix”
- Cabe ao projetista decidir, considerando...
  - Desempenho
  - Comportamento esperado

8.2 Arquitetura de um SAD

- Uma arquitetura que ofereça uma separação clara entre preocupações, estruturando o acesso a arquivos através de três componentes:
  - Serviço de arquivos simples ou individuais
  - Um serviço de diretório
  - Um módulo cliente
- O módulo cliente usa as interfaces do serviço **de arquivos simples** e de **diretório.** É invocado localmente e repassa a solicitação para servidores remotos.

Arquitetura de um Serviço de Arquivos

Figure 8.5



*

Responsabilidades dos vários módulos

- Serviço de arquivos simples:
  - Preocupado com a implementação de operações no conteúdo dos arquivos. *Unique File Identifiers* (UFIDs) são usados para referenciar a arquivos em todas as requisições. UFIDs são longas sequências de bits únicas para cada arquivo no SAD.
- Serviço de Diretório:
  - Provê mapeamento entre nomes textuais e seus UFIDs. Programas clientes podem obter o UFID de um arquivo buscando pelo nome no serviço de diretório. Suporta diferentes funções necessárias como criar arquivos, remover nomes de arquivos de diretórios, etc.
- Módulo Cliente:
  - Roda em cada computador cliente e provê acesso a ambos os serviços como uma única API.
  - Mantém informações sobre localizações dos processos que oferecem ambos os serviços na rede; Permite ganho de desempenho através da implementação de cache de blocos de arquivos recentemente usados por um cliente.

Operações disponibilizadas pelos serviços

Figuras 8.6 e 8.7

Serviço de arquivos simples

Posição do 1o byte

Read(fileId, i, n) -> Data

Posição do 1o byte

Write(FileId, i, Data)

Create() -> FileId

Delete(FileId)

GetAttributes(FileId) -> Attr

SetAttributes(FileId, Attr)

Serviço de diretório

Lookup(Dir, Name) -> FileId

AddName(Dir, Name, ~~File~~) FileId

UnName(Dir, Name)

GetNames(Dir, Pattern) -> NameSeq

FileId

Identificador único para arquivos no SAD. Similar à referência de objeto remoto descrito no capítulo 4

*

Operações disponibilizadas pelos serviços

Figures 8.6 and 8.7

Serviço de arquivos simples

Posição do 1o byte

Read(fileId, i, n) -> Data

position of first byte

Write(FileId, i, Data)

Create() -> FileId

Delete(FileId)

GetAttributes(FileId) -> Attr

SetAttributes(FileId, Attr)

Serviço de diretório

Lookup(Dir, Name) -> FileId

AddName(Dir, Name, ~~File~~) FileId

UnName(Dir, Name)

GetNames(Dir, Pattern) -> NameSeq

Busca de diretórios

Diretórios como '/usr/bin/tar' são identificados através de chamadas recursivas ao método *lookup()*, cada chamada busca um componente do diretório, iniciando com o ID do diretório raiz '/' conhecido por qualquer cliente.

*

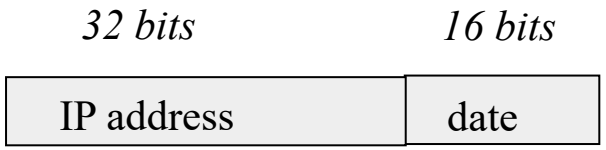
Grupo de Arquivos – File Groups

Uma coleção de arquivos que podem ser localizados em qualquer servidor ou movido entre servidores mantendo os mesmos nomes.

- Similar ao *filesystem* UNIX (!= de partição)
- Ajuda distribuir a carga entre diversos servidores .
- Grupos de arquivos tem identificadores únicos no sistema de forma global

Para construir um ID globalmente unico, usa-se algum atributo da máquina onde ele estiver criado como o IP, mesmo que o grupo se mova posteriormente

File Group ID:



*

Replicação em Sistemas de Arquivos

- SDs geralmente oferecem alguma forma de replicação de dados
  - Múltiplas cópias do mesmo objeto
- Por que?
  - Aumentar confiabilidade (Tolerância a falhas)
  - Permitir acesso mesmo na caso de falha
  - Divisão de carga entre servidores
- Principal problema: transparência (!)

Replicação em Sistemas de Arquivos

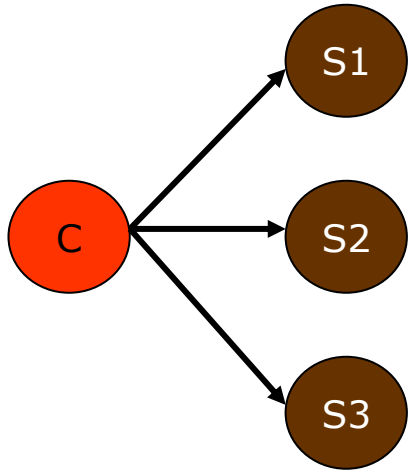
- Quanto o usuário sabe da replicação?
  - Sabe e pode até manipular
  - Sistema faz tudo, sozinho → transparente
- Replicação pode ser feita, basicamente, de três formas
  - Replicação explícita
  - Replicação atrasada
  - Replicação usando grupo

Replicação em Sistemas de Arquivos

- Quanto o usuário sabe da replicação?
  - Sabe e pode até manipular
  - Sistema faz tudo, sozinho → transparente
- Replicação pode ser feita, basicamente, de três formas
  - Replicação explícita
  - Replicação atrasada
  - Replicação usando grupo

Replicação explícita

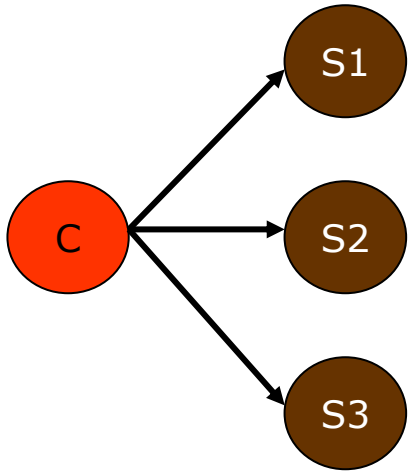
- Programador faz todo o trabalho sujo
- Serviço de diretório pode permitir múltiplos UFIDs por arquivo
  - Recupera todos no lookup
  - Quando for manipular, tenta seqüêncialmente um por um dos UFIDs, até conseguir
- Funciona, mas é muito trabalhoso, e nada transparente



Arquivo1	1.14	2.41	3.56
Prog.c	1.32	2.56	3.23

Replicação explícita

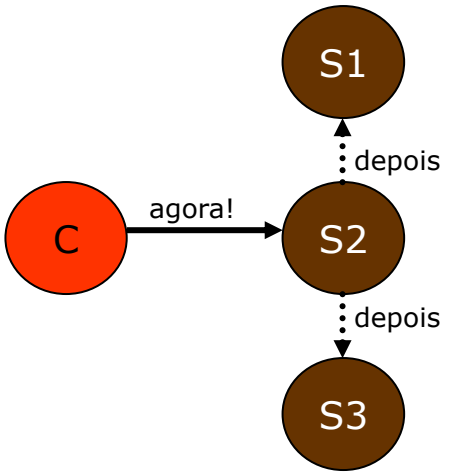
- Programador faz todo o trabalho sujo
- Serviço de diretório pode permitir múltiplos FIDs por arquivo
  - Recupera todos no lookup
  - Quando for manipular, tenta seqüêncialmente um por um dos FIDs, até conseguir
- Funciona, mas é muito trabalhoso, e nada transparente



Arquivo1	1.14	2.41	3.56
Prog.c	1.32	2.56	3.23

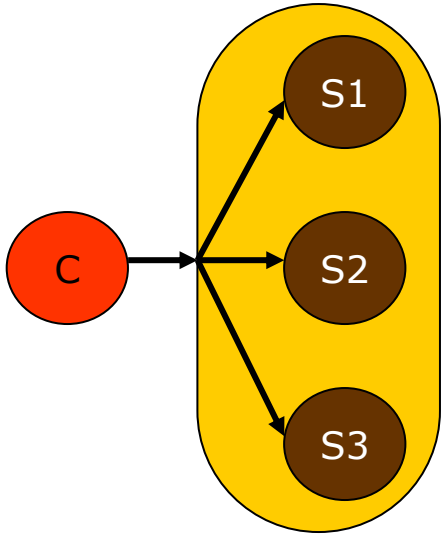
Replicação atrasada

- Apenas uma cópia é feita em um servidor
- Servidor é responsável por...
  - Propagar atualizações
  - Selecionar outro servidor para atender, se ele não puder



Replicação usando grupo

- Operação de escrita é feita ao mesmo tempo em todos
  - Multicast atômico
  - Mantém todos atualizados em caso de escrita
  - Problema: confiabilidade



Protocolos de atualização – update de arquivos

- Até agora, só resolvemos o processo de criação
- Atualização não é tão simples
  - Mensagem de update para cada cópia
  - Se o processo coordenador falhar, cópias não alteradas → inconsistências
- Duas formas
  - Replicação de cópia primária
  - Algoritmos de votação (abordado em uma outra aula)

Replicação de cópia primária

- Um servidor é eleito primário
  - Outros são secundários
- Atualização
  - Entregue para o primário
  - Atualizada cópia local
  - Grava em memória estável
  - Notifica secundários

Replicação de cópia primária

- No caso de falha
  - Uma das réplicas entra no ar – assumindo o papel do primário
    - *Pode ser utilizado algum algoritmo de votação entre as réplicas*
  - Quando o primário voltar a funcionar ele entrará em funcionamento como uma réplica e receberá as atualizações perdidas (sincronismo)
  - Ele pode ser promovido a primário novamente ou permanecer como secundário/réplica
- Leitura pode ser feita de qualquer um
- Problema
  - Ponto único de falha
    - *Tempo para réplica assumir pode ser alto, gerando uma falta momentânea de sistema*

DFS: Estudos de Caso

- NFS (Network File System)
- OCEANSTORE
- FARSITE
- AFS - pesquisa bibliográfica)
- HADOOP - pesquisa bibliográfica

NFS

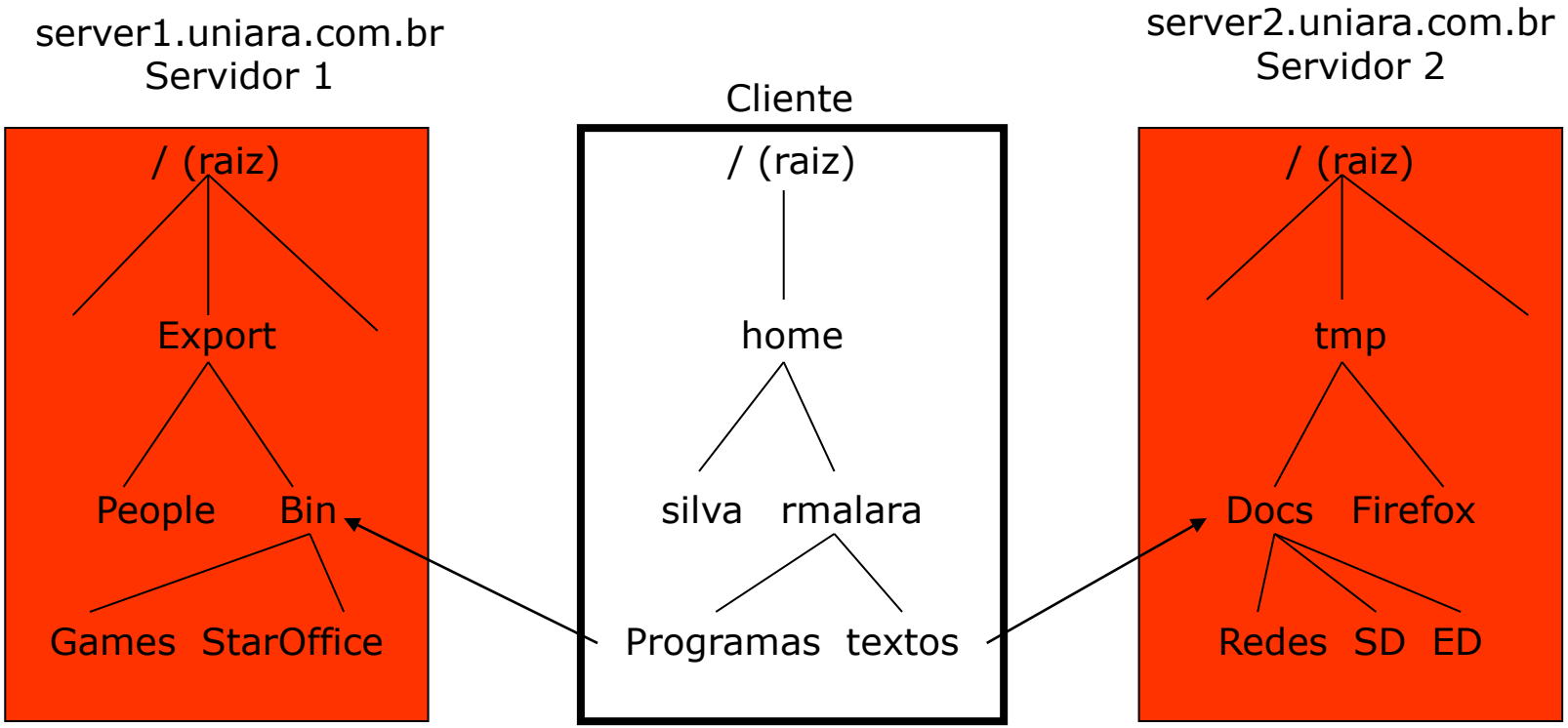
- Network File System
- Desenvolvido pela SUN
  - Clientes/servidores em quase todos os SOs
  - Permite operação heterogênea
    - *Clientes Windows, servidores UNIX e DOS*
- Interessante observar
  - Arquitetura
  - Protocolos
  - Implementação

Arquitetura NFS

- Idéia básica
  - Coleção clientes e servidores terem acesso a arquivos compartilhados
  - Participantes podem estar numa LAN ou WAN
- Máquinas podem ter processo C/S ao mesmo tempo
- Servidores exportam diretórios e clientes os montam

Arquitetura NFS

- Montagem do diretório
  - Ponto de montagem é livre



Arquitetura NFS

- Montagem estática ou dinâmica
  - Dinâmica introduz alguma tolerância a falha
  - Várias entradas para mesmo ponto de montagem
- Dois clientes montam o mesmo diretório
  - Eles podem se comunicar através de arquivos desse diretório
- Simplicidade é o principal atrativo
- Como NFS se comporta em relação a transparência?

Transparências no NFS

- Acesso
  - Clientes manipulam arquivos remotos da mesma forma que arquivos locais
- Localização
  - No padrão, cada um monta como quiser (ruim)
  - Configuração das tabelas de exportação e pontos de montagem podem garantir essa transparência (NIS – Network Information Service)
  - E.g., /home/malara (onde está /home???)

Transparências no NFS

- Falha
  - NFS é stateless e maior parte das operações é idempotente (lembram ?)
  - Se servidor falhar, cliente sabe onde parou
  - Se cliente falha, servidor não tem estado para limpar
- Desempenho
  - Técnicas de cache
  - Aumento médio de 20% no tempo de acesso

Transparências no NFS

- Migração
  - Se diretório exportado mudou de servidor, alterar tabela de pontos de montagem nos clientes

- Nem tudo é perfeito...

”Opacidades” do NFS

- Replicação
  - NFS *per si* não faz replicação
  - Network Information Service – NIS porém ele não foi feito para isso
    - *Replicação de cópia primária de alguns arquivos (ex: arquivos de configuração)*
- Concorrência
  - Apenas o suporte oferecido pelo SO (e.g., lock)
- Escalabilidade
  - Projeto original: 10 clientes por servidor
  - Servidor se torna gargalo do sistema

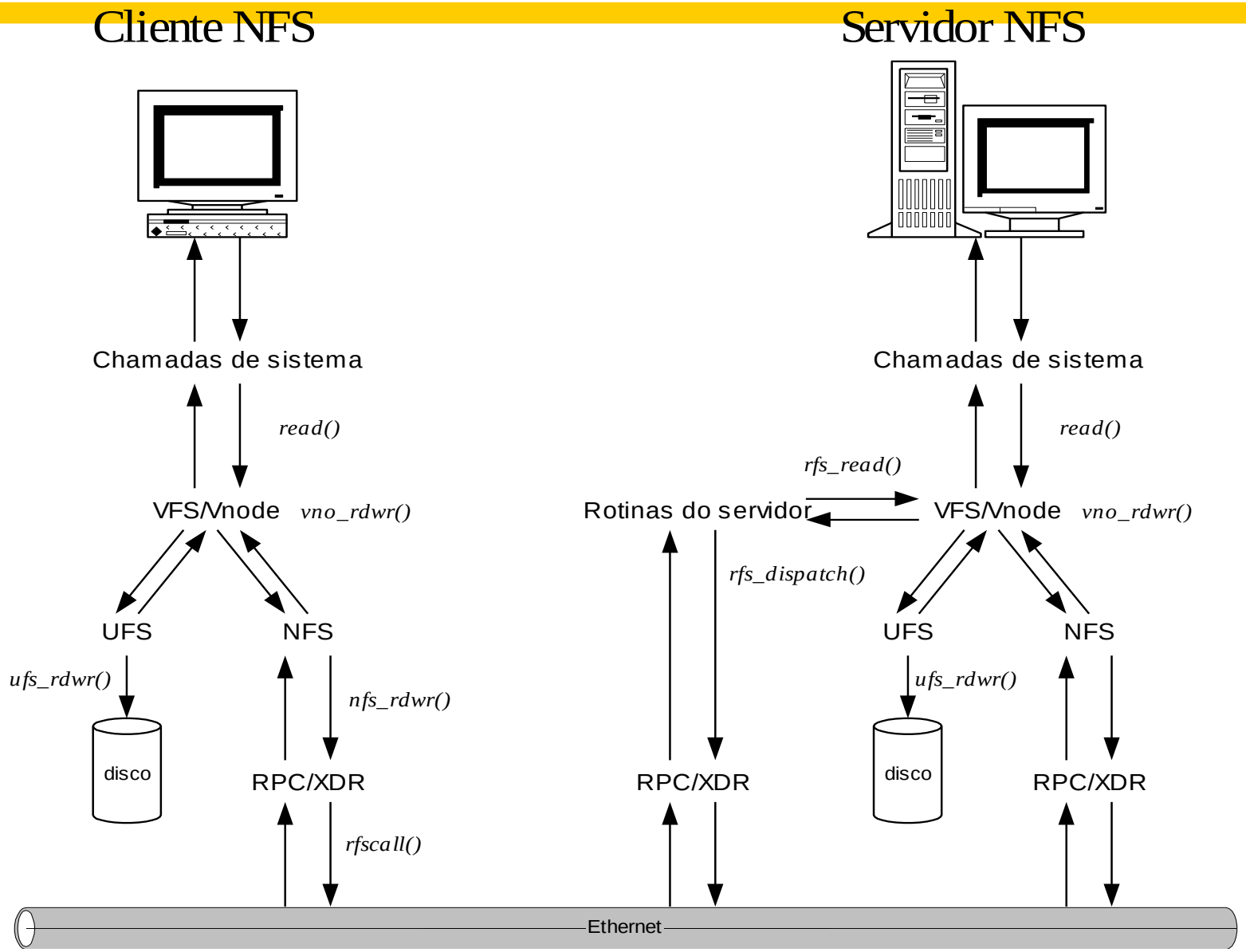
Protocolos

- NFS usa RPC
- Aceita grande parte das chamadas UNIX
  - Exceto OPEN e CLOSE
  - Servidor é stateless
- Semântica similar ao UNIX
  - Problema: não tem estado → o que está aberto?

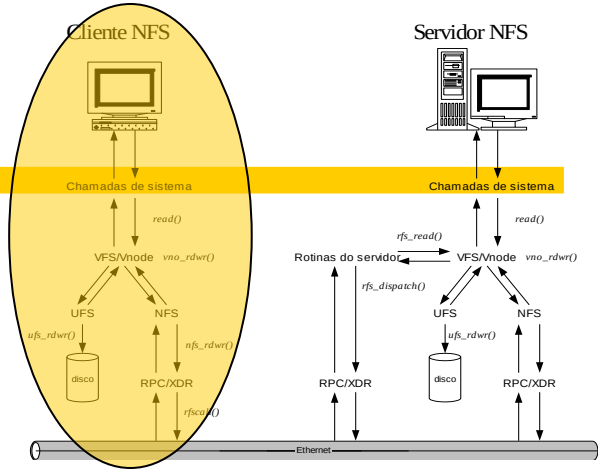
Protocolos

- NFS usa operações read-ahead (leitura adiantada)/ delayed-write (escrita atrasada)
- Como resolver consistência de cache (sim, usa-se cache) ?
  - Temporizadores
    - *3 e 30seg para arquivos e diretórios*
    - *Cópia do cache descartada quanto expira*
  - Sempre que consultar cache, consultar servidor
    - *Se local for a mais nova, ok!*
    - *C.c., recuperar última do servidor*

Implementação

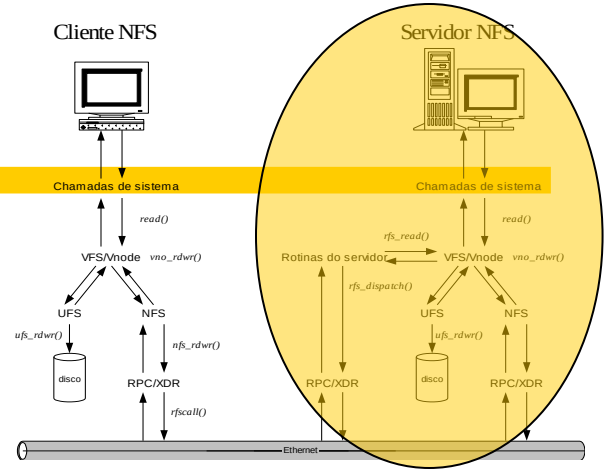


Implementação



- Virtual File System
  - Usa nós-v
    - *Um nó-v aponta para um nó-r ou um nó-i*
- Cliente faz uma requisição de montagem
  - Monta no VFS um nó-v apontando para um nó-r
- Cliente quer ler
  - VFS percebe que nó-v aponta para nó-r
  - Emite RCP para o servidor
  - Repassa para o cliente

Implementação



- Servidor recebe requisição remota
  - Para onde aponta nó-v local
    - *Nó-r → requisição é passada para o servidor correto*
    - *Nó-i → atende localmente através do User File System*

- Servidores podem exportar diretórios remotos
  - Cascata de requisições

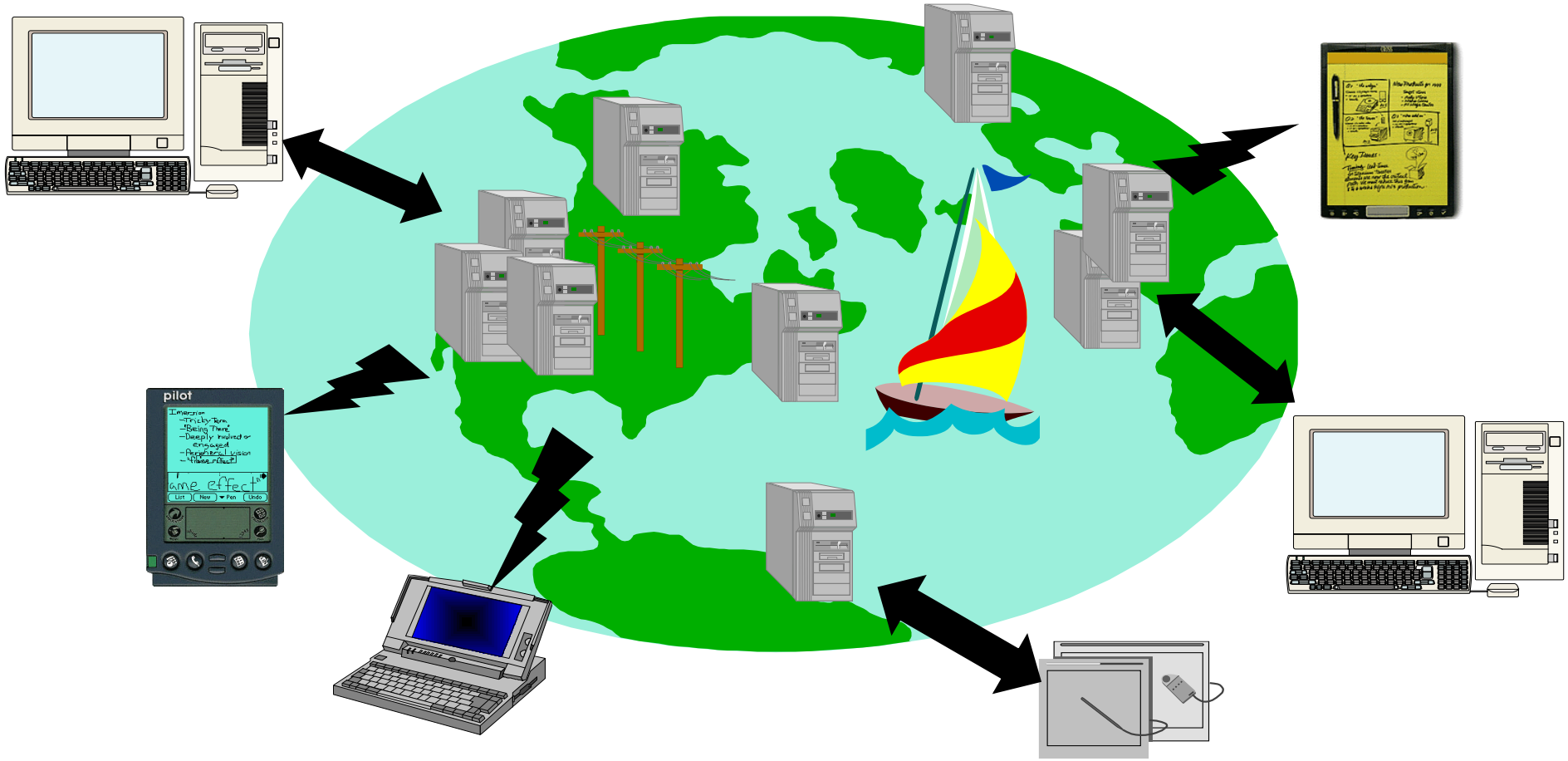
NFS

- Padrão *de fato* para compartilhamento
- Versões para virtualmente qualquer SO
  - Integração de HW e SW diferentes

OceanStore

- Sistema de armazenamento persistente de escala global
  - Servidores não-confiáveis
  - Dados nômades
- Capaz de gerenciar até 10^{14} arquivos
- Usado para computação ubíqua
  - Computador em todo o lugar
  - Relógio de pulso não tem capacidade de armazenamento → OceanStore
- http://oceanstore.cs.berkeley.edu/

OceanStore

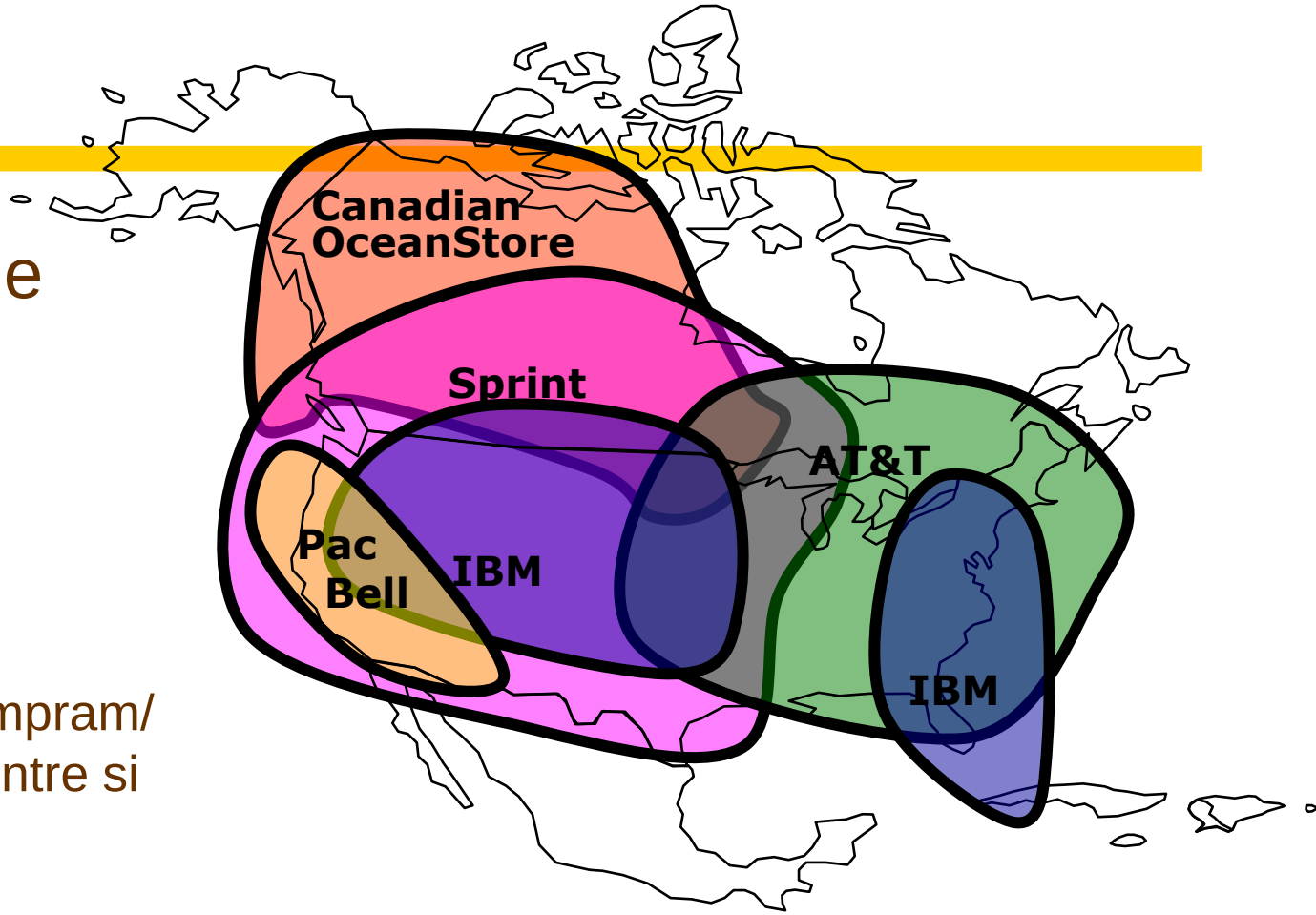


OceanStore

- Onde a informação está?
  - Geograficamente distribuída → disponibilidade
- Como está protegida?
  - Criptografia e assinatura digital
- Como ela é indestrutível
  - Redundância com redistribuição e reconstrução
- Como é gerenciada?
  - Sozinha!
- Quem oferece espaço
  - Infraestrutura de utilidade (provedores de espaço)

OceanStore

- Infraestrutura de utilidade
  - Federações
  - Taxa mensal paga a um forne-cedor
  - Fornecedores compram/ vendem espaço entre si



OceanStore assume que...

- Infraestrutura não confiável
  - Apenas conteúdo cifrado no OceanStore
- Alguém é responsável
  - Pela durabilidade e consistência, sem garantir privacidade
- Grande parte está bem conectada
  - Rede de alto desempenho entre produtor/ consumidor
  - Usar multicast para consistência quando puder

Como está o OceanStore

- “Irá o lago se transformar em oceano ?”
  - Replicando
  - Localizando
  - Fazendo manutenção
  - Fragmentando
- Parte do projeto Endevour
  - Tornar a computação mais acessível
- http://oceanstore.cs.berkeley.edu/

Farsite

- Federated, Available, and Reliable Storage for an Incompletely Trusted Environment
  - Sistema de arquivos distribuído sem servidor
  - Todas as estações cooperam no armazenamento
- Aspecto chave
  - Participantes não são confiáveis
    - *Podem ser formatados*
    - *Podem ser invadidos e dados corrompidos*

Farsite

- Tudo o que o Farsite oferece pode ser obtido com servidor central, mas...
  - E/S de alto desempenho
  - Matrizes de discos
  - Manutenção
  - Gargalo
  - Ponto central de falha

Farsite

- Clientes cooperam e definem
  - Quem pode armazenar
  - Como dividir o espaço
- Várias réplicas
  - Fragmentação
  - Aumento do espaço necessário (redundância)

Farsite

- Ainda sem implementação real
- Alguns estudos mostram que...
  - Cada máquina serve 10Mb/dia em cache
  - Cada máquina recebe tráfego médio de 7Mb/hora/réplica
  - Razoável e manipulável pelos PCs de hoje
- Desenvolvido na Microsoft
  - http://research.microsoft.com/en-us/projects/farsite/

OceanStore e Farsite

- Exploram Lei de Moore
  - Capacidade de armazenamento crescente
  - Custo decrescente
  - Largura de banda da rede aumentando
  - Aumento de desempenho das estações
- Diferença básica
  - Escala (global x regional)

DFS: Case Studies

- NFS (Network File System)
  - Developed by Sun Microsystems (in 1985)
  - Most popular, open, and widely used.
  - NFS protocol standardised through IETF (RFC 1813)
- AFS (Andrew File System)
  - Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)
  - A research project to create campus wide file system.
  - Public domain implementation is available on Linux (LinuxAFS)
  - It was adopted as a basis for the DCE/DFS file system in the Open Software Foundation (OSF, www.opengroup.org) DEC (Distributed Computing Environment)

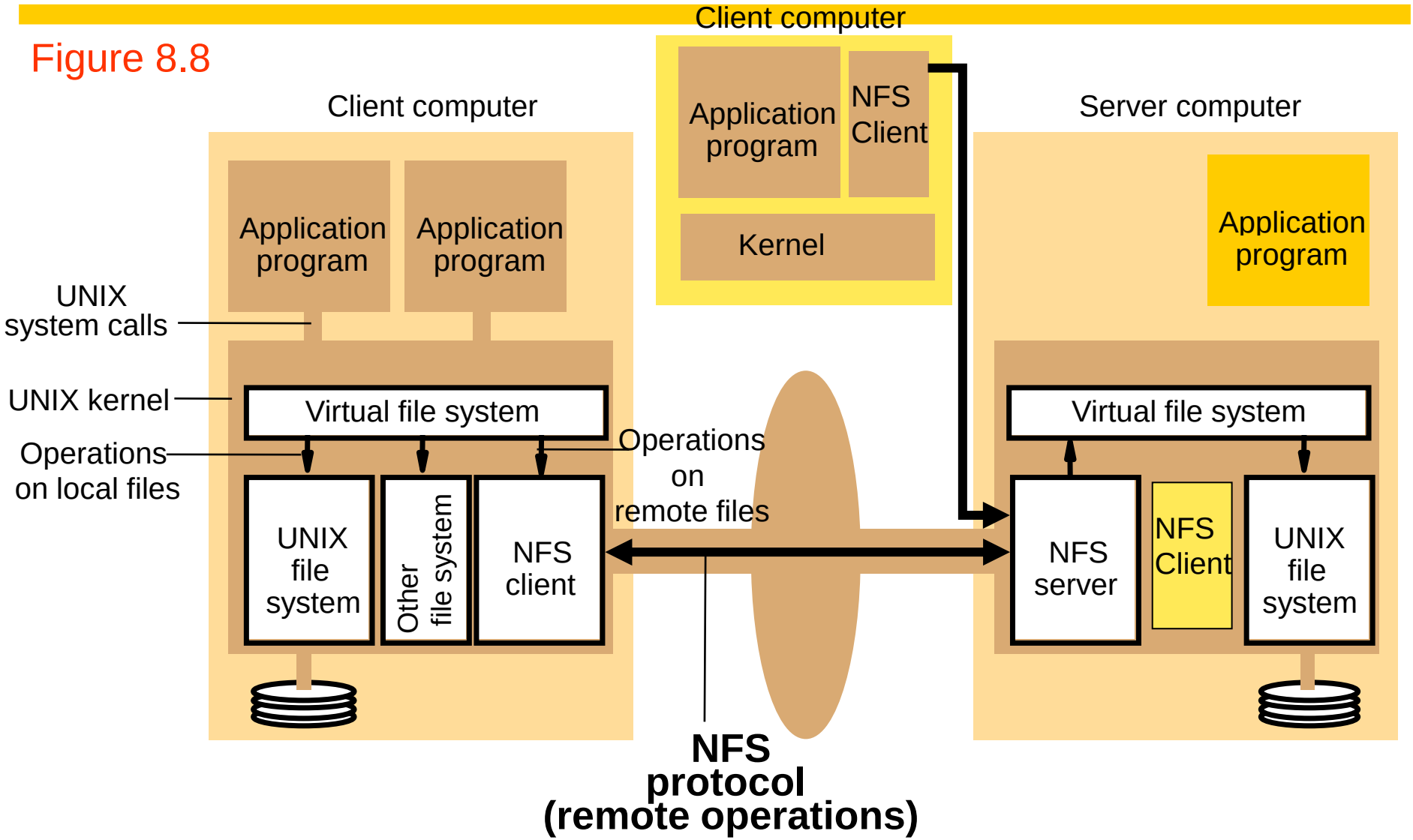
Case Study: Sun NFS

- An industry standard for file sharing on local networks since the 1980s
- An open standard with clear and simple interfaces
- Closely follows the abstract file service model defined above
- Supports many of the design requirements already mentioned:
  - transparency
  - heterogeneity
  - efficiency
  - fault tolerance
- Limited achievement of:
  - concurrency
  - replication
  - consistency
  - security

*

NFS architecture

Figure 8.8



NFS architecture: does the implementation have to be in the system kernel?

No:

- there are examples of NFS clients and servers that run at application-level as libraries or processes (e.g. early Windows and MacOS implementations, current PocketPC, etc.)

But, for a Unix implementation there are advantages:

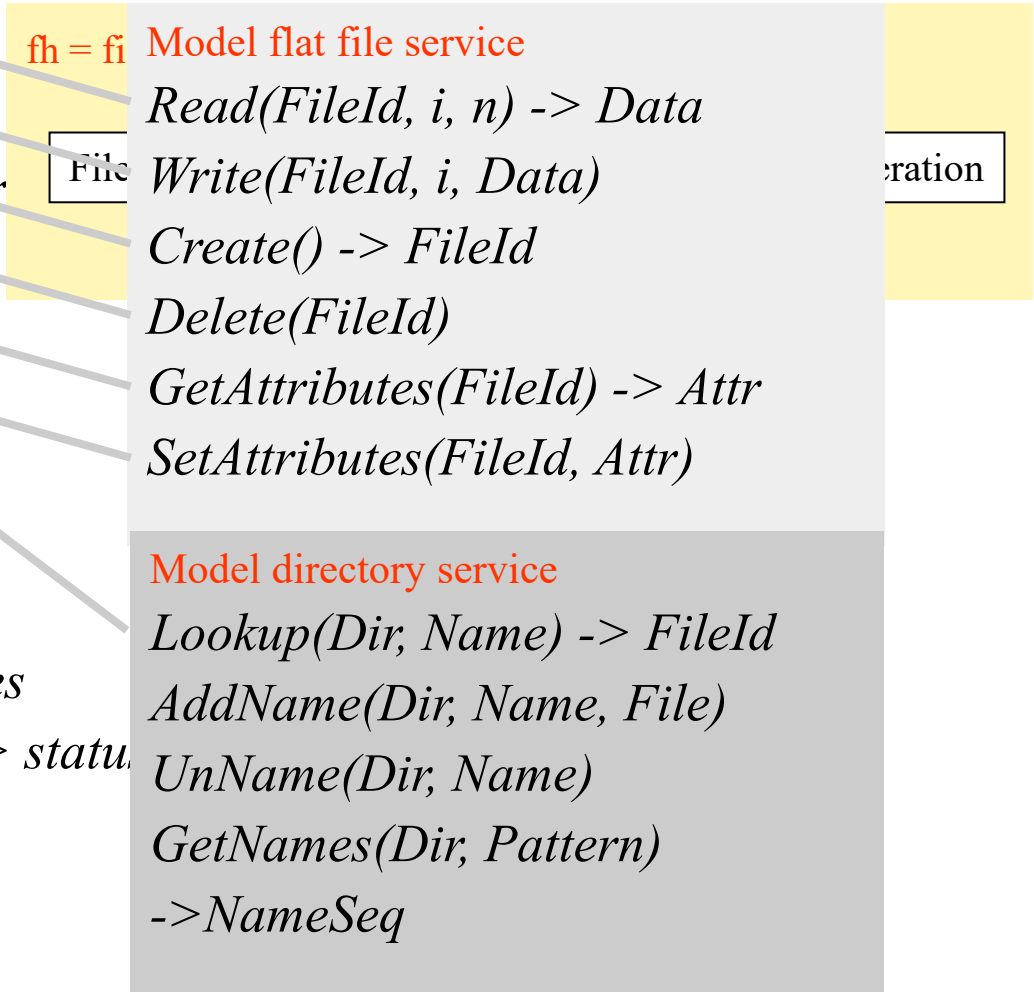
- Binary code compatible - no need to recompile applications
  - *Standard system calls that access remote files can be routed through the NFS client module by the kernel*
- Shared cache of recently-used blocks at client
- Kernel-level server can access i-nodes and file blocks directly
  - *but a privileged (root) application program could do almost the same.*
- Security of the encryption key used for authentication.

*

NFS server operations (simplified)

Figure 8.9

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> statu*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*



*

NFS access control and authentication

- Stateless server, so the user's identity and access rights must be checked by the server on each request.
  - In the local file system they are checked only on *open()*
- Every client request is accompanied by the userID and groupID
  - not shown in the Figure 8.9 because they are inserted by the RPC system
- Server is exposed to imposter attacks unless the userID and groupID are protected by encryption
- Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution
  - Kerberos is described in Chapter 7.

*

Mount service

- Mount operation:

  mount(remotehost, remotedirectory, localdirectory)

- Server maintains a table of clients who have mounted filesystems at that server
- Each client maintains a table of mounted file systems holding:

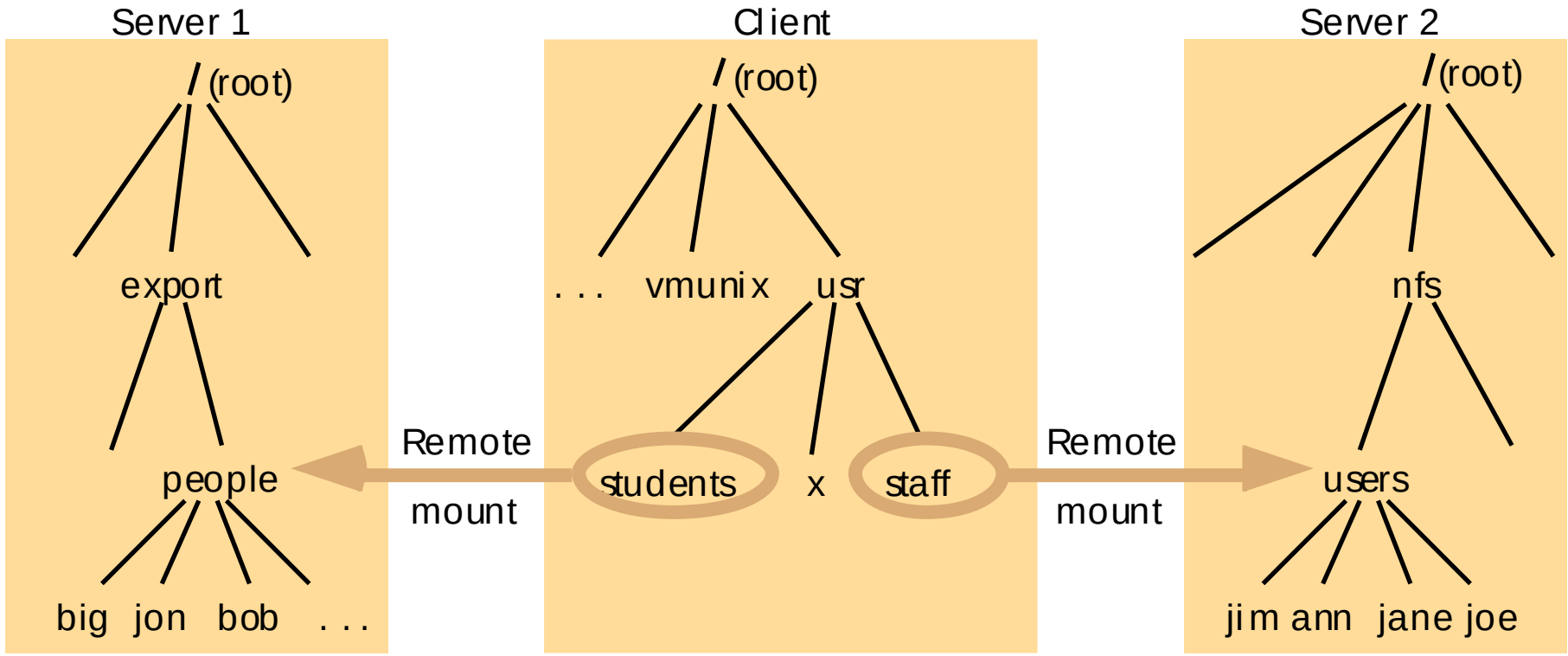
  < IP address, port number, file handle>

- *Hard* versus *soft* mounts

*

Local and remote file systems accessible on an NFS client

Figure 8.10



Note: The file system mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server 1; the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

*

NFS optimization - server caching

- Similar to UNIX file caching for local files:
  - pages (blocks) from disk are held in a main memory buffer cache until the space is required for newer pages. Read-ahead and delayed-write optimizations.
  - For local files, writes are deferred to next sync event (30 second intervals)
  - Works well in local context, where files are always accessed through the local cache, but in the remote case it doesn't offer necessary synchronization guarantees to clients.
- NFS v3 servers offers two strategies for updating the disk:
  - *write-through* - altered pages are written to disk as soon as they are received at the server. When a *write()* RPC returns, the NFS client knows that the page is on the disk.
  - *delayed commit* - pages are held only in the cache until a *commit()* call is received for the relevant file. This is the default mode used by NFS v3 clients. A *commit()* is issued by the client whenever a file is closed.

*

NFS optimization - client caching

- Server caching does nothing to reduce RPC traffic between client and server
  - further optimization is essential to reduce server load in large networks
  - NFS client module caches the results of *read*, *write*, *getattr*, *lookup* and *readdir* operations
  - synchronization of file contents (*one-copy semantics*) is not guaranteed when two or more clients are sharing the same file.
- Timestamp-based validity check
  - reduces inconsistency, but doesn't eliminate it
  - validity condition for cache entries at the client:

    $(T - Tc < t) \vee (Tm_{client} = Tm_{server})$
  - t is configurable (per file) but is typically set to 3 seconds for files and 30 secs. for directories
  - it remains difficult to write distributed applications that share files with NFS

t	freshness guarantee
Tc	time when cache entry was last validated
Tm	time when block was last updated at server
T	current time

*

NFS performance

- Early measurements (1987) established that:
  - *write()* operations are responsible for only 5% of server calls in typical UNIX environments
    - *hence write-through at server is acceptable*
  - *lookup()* accounts for 50% of operations -due to step-by-step pathname resolution necessitated by the naming and mounting semantics.
- More recent measurements (1993) show high performance:

  1 x 450 MHz Pentium III: > 5000 server ops/sec, < 4 millisec. average latency

  24 x 450 MHz IBM RS64: > 29,000 server ops/sec, < 4 millisec. average latency

  see www.spec.org for more recent measurements
- Provides a good solution for many environments including:
  - large networks of UNIX and PC clients
  - multiple web server installations sharing a single file store

*

Recent advances in file services

NFS enhancements

WebNFS - NFS server implements a web-like service on a well-known port. Requests use a 'public file handle' and a pathname-capable variant of *lookup().* Enables applications to access NFS servers directly, e.g. to read a portion of a large file.

One-copy update semantics (Spritely NFS, NQNFS) - Include an *open()* operation and maintain tables of open files at servers, which are used to prevent multiple writers and to generate callbacks to clients notifying them of updates. Performance was improved by reduction in *gettattr()* traffic.

Improvements in disk storage organisation

RAID - improves performance and reliability by striping data redundantly across several disk drives

Log-structured file storage - updated pages are stored contiguously in memory and committed to disk in large contiguous blocks (~ 1 Mbyte). File maps are modified whenever an update occurs. Garbage collection to recover disk space.

*

New design approaches 1

Distribute file data across several servers

- Exploits high-speed networks (ATM, Gigabit Ethernet)
- Layered approach, lowest level is like a 'distributed virtual disk'
- Achieves scalability even for a single heavily-used file

'Serverless' architecture

- Exploits processing and disk resources in all available network nodes
- Service is distributed at the level of individual files

Examples:

xFS (section 8.5): Experimental implementation demonstrated a substantial performance gain over NFS and AFS

Frangipani (section 8.5): Performance similar to local UNIX file access

Tiger Video File System (see Chapter 15)

Peer-to-peer systems: Napster, OceanStore (UCB), Farsite (MSR), Publius (AT&T research) - see web for documentation on these very recent systems

*

New design approaches 2

- Replicated read-write files
  - High availability
  - Disconnected working
    - *re-integration after disconnection is a major problem if conflicting updates have ocurred*
  - Examples:
    - *Bayou system (Section 14.4.2)*
    - *Coda system (Section 14.4.3)*

*

Summary

- Distributed File systems provide illusion of a local file system and hide complexity from end users.
- Sun NFS is an excellent example of a distributed service designed to meet many important design requirements
- Effective client caching can produce file service performance equal to or better than local file systems
- Consistency *versus* update semantics *versus* fault tolerance remains an issue
- Most client and server failures can be masked
- Superior scalability can be achieved with whole-file serving (Andrew FS) or the distributed virtual disk approach

Future requirements:

- support for mobile users, disconnected operation, automatic re-integration (Cf. Coda file system, Chapter 14)
- support for data streaming and quality of service (Cf. Tiger file system, Chapter 15)

*

Exercise A solution

Write a simple C program to copy a file using the UNIX file system operations shown in Figure 8.4.

```
#define BUFSIZE 1024
#define READ 0
#define FILEMODE 0644
void copyfile(char* oldfile, char* newfile)
{       char buf[BUFSIZE]; int i,n=1, fdold, fdnew;

        if((fdold = open(oldfile, READ))>=0) {
                fdnew = creat(newfile, FILEMODE);
                while (n>0) {
                        n = read(fdold, buf, BUFSIZE);
                        if(write(fdnew, buf, n) < 0) break;
                }
        close(fdold);   close(fdnew);
        }
        else printf("Copyfile: couldn't open file: %s \n", oldfile);
}
main(int argc, char **argv)  {
        copyfile(argv[1], argv[2]);
}
```


*

Exercise B solution

Show how each file operation of the pro
using the operations of the Model File S

```
if((fdold = open(oldfile, READ))>=0) {
        fdnew = creat(newfile, FILEMODE);
        while (n>0) {
                n = read(fdold, buf, BUFSIZE);
                if(write(fdnew, buf, n) < 0) breal
        }
close(fdold);    close(fdnew);
```

server operations for: copyfile("/usr/include/glob.h", "/foo")

fdold = open('/usr/include/glob.h", READ)

Client module actions:

FileId = Lookup(Root, "usr") - *remote invocation*
FileId = Lookup(FileId, "include") - *remote invocation*
FileId = Lookup(FileId, "glob.h") - *remote invocation*

client module makes an entry in an *open files* table with *file = FileId, mode = READ,* and *RWpointer* = 0. It returns the table row number as the value for *fdold*

fdnew = creat("/foo", FILEMODE)

Client module actions:

FileId = create() - *remote invocation*
AddName(Root, "foo", FileId) - *remote invocation*
SetAttributes(FileId, attributes) - *remote invocation*

client module makes an entry in its *openfiles* table with *file = FileId, mode = WRITE,* and *RWpointer* = 0. It returns

n = read(fdold, buf, BUFSIZE)

Client module actions:

Read(openfiles[fdold].file, openfiles[fdold].RWpointer, BUFSIZE)
- *remote invocation*

increment the *RWpointer* in the *openfiles* table by BUFSIZE and assign the resulting array of data to *buf*

*