CONSIDERAÇÕES PARA O DESENVOLVIMENTO DE APLICAÇÕES DISTRIBUÍDAS EM AMBIENTES HETEROGÊNEOS

Gedeon J. dos Santos Filho
Infocon Tecnologia Ltda. - Rua Manoel Barros de Oliveira, 303 - Bodocongó
CEP 58.109-125 Campina Grande - PB - Brasil - Fone: (083) 333-1904 - Fax: (083) 333-1528

*Jacques P. Sauvé**

*J. Antão B. Moura**
*UFPB/CCT/DSC - Av. Aprigio Veloso S/N - Bodocongó
CEP 58.100 - Campina Grande - PB - Brasil
e-mail: antao@dsc.ufpb.br

Resumo

Este trabalho apresenta considerações práticas para orientar o programador no desenvolvimento de Aplicações Distribuídas em sistemas de redes heterogêneas. Após uma breve discussão da arquitetura e serviços de um Ambiente de Computação Distribuida, discutem-se os principais requisitos para o desenvolvimento de Aplicações Distribuidas, seguido de considerações específicas para projeto, tais como sistemas de arquivos distribuídos, seleção de protocolo de transporte, tratamento e recuperação de erros e desempenho.

Abstract

This work presents practical considerations for developers of applications running on heterogeneous distributed systems. After briefly exploring the Architecture and Services of distributed environments, the paper focus on the requirements of Distributed Applications in order to set the proper framework for discussing recommendations concerning distributed file systems, transport protocol, error handling and performance issues.

Palavras-chaves: *Computação Distribuída. Sistemas Distribuídos Heterogêneos. Desenvolvimento de Aplicações. Sistemas de Arquivos Distribuído. Protocolo de Transporte. Tratamento e Recuperação de Erros. Desempenho.*

Key-words: *Distributed Computing. Heterogeneous Distributed Systems. Application Development. Distributed File Systems. Transport Protocol. Error Handling. Performance.*

1 Introdução

O interesse por sistemas de computação distribuída vem crescendo rapidamente nos últimos anos. A redução dos custos de *hardware*, combinada com os avanços tecnológicos dos sistemas de redes de computadores, tem viabilizado o *downsizing*, estimulando a utilização de Aplicações Distribuídas (**ADs**).

Uma aplicação é considerada distribuída se seus processos podem ser executados paralelamente em espaços de endereçamento de diferentes computadores que estão conectados através de uma rede de comunicação. Os processos que residem em diferentes máquinas são classificados como processos remotos. Quando os processos podem residir em computadores com arquitetura ou sistema operacional distintos, então a AD é heterogênea.

As ADs heterogêneas são um meio que o programador dispõe para ter acesso racional aos recursos[1] existentes nos sistemas de redes heterogêneas. Suas principais vantagens são: confiabilidade, alto desempenho, uso especializado de *hardware* e modularidade.

• **Confiabilidade**: as ADs possuem um potencial inerente de tolerância a falhas. Se um computador que contém processos da aplicação falha[2], qualquer outro conectado à rede pode ser usado para assumir as tarefas pendentes.

• **Alto Desempenho**: as aplicações que fazem uso intenso de CPU podem ser projetadas de forma a aproveitarem o potencial das múltiplas CPUs (e outros recursos) disponíveis na rede. Os processos remotos das ADs permitem ao programador obter paralelismo sem a necessidade do uso de *hardware* multiprocessador. Por exemplo, aplicações para processamento de imagens ficam extremamente mais rápidas com o processamento de partes das imagens em diferentes computadores da rede [1].

• **Uso Especializado do Hardware**: o fato dos processos das ADs poderem ser executados em diferentes computadores, permite, por exemplo, que se desenvolva uma aplicação que consulte um banco de dados de um *mainframe*, processe os dados obtidos em um supercomputador e, finalmente, mostre os resultados em forma de gráfico em uma estação de trabalho. Isso permite a racionalização de uso dos recursos existentes na rede, proporcionando uma melhor relação custo/desempenho.

• **Modularidade**: o fato das ADs serem compostas por um conjunto de processos cooperativos, normalmente executados em diferentes computadores, resulta em uma natureza modular intrínseca. Em uma aplicação centralizada, a modularidade é um conceito lógico. Em uma AD, a modularidade é física. Essa modularidade inerente pode ser usada em benefício do programador, por oferecer maior flexibilidade para adicionar, eliminar e modificar os módulos da AD.

Este trabalho fornece considerações para subsidiar decisões em projetos de ADs. As considerações oferecidas resultam de experiências práticas de desenvolvimento de ADs na UFPB e em empresas de *software* de Campina Grande - PB. Assim, espera-se contribuir para estender e complementar o acesso bibliográfico sobre projeto, desenvolvimento e testes de aplicações em sistemas distribuídos heterogêneos.

O restante do trabalho é organizado em mais quatro seções. As seções 2 e 3 discutem a infra-estrutura e os requisitos básicos necessários para o desenvolvimento de ADs respectivamente. A seção 4 apresenta considerações específicas para projeto de ADs envolvendo: seleção de protocolo de transporte, tratamento e recuperação de erros e desempenho. Finalmente, a seção 5 é dedicada a conclusões.

---

[1] Computadores, processos, unidades de disco e fitas, impressoras, arquivos, bancos de dados, *plotters*, etc.

[2] Neste trabalho, falha é todo evento causado por um defeito de *hardware* ou *software*, que viola a especificação da AD, dando origem a um erro.

2 Infra-Estrutura para Aplicações Distribuídas

Idealmente, para desenvolver ADs é necessário um conjunto integrado de serviços a fim de evitar que o programador tenha que tratar, a nível de aplicação, toda a complexidade de um sistema de redes heterogêneas. Os Ambientes de Computação Distribuída Heterogênea (**ACDs**) atendem a essa necessidade.

Um ACD pode ser definido como um modelo arquitetural de computação no qual uma coleção de processos, recursos e computadores de diferentes fabricantes, distribuídos numa rede, trabalham cooperativamente para executar tarefas [2].

2.1 Arquitetura e Serviços

Os ACDs possuem uma arquitetura em camadas composta por um conjunto integrado de serviços (**Fig. 1** [3]). A arquitetura do ACD procura tornar transparente a complexidade do sistema de rede para o usuário final, para o programador e para o administrador do sistema . Do ponto de vista das aplicações, o ACD pode ser visto tanto como um único sistema lógico quanto como uma coleção de serviços independentes, que podem ser utilizados individualmente ou combinados.

É de interesse aqui, o exame dos serviços de ACDs do ponto de vista de aplicações. O exame é iniciado com a breve apresentação de cada serviço nas nove subseções que seguem.

2.1.1 Serviço de Transporte

A função do serviço de transporte é integrar as diferentes tecnologias das redes físicas existentes no ACD, formando uma única rede lógica. Para tornar possível um trabalho cooperativo, face às diferentes tecnologias existentes, o serviço de transporte pode fazer uso praticamente de qualquer pilha de protocolos, como por exemplo, TCP/IP, SNA ou OSI [4] [5] [6] [7].

2.1.2 Serviço de Threads

Um *thread* representa o agente de controle de uma sequência de instruções sendo executadas por um programa. A partir do serviço de *threads*, um único processo pode possuir vários *threads* de controle, o que permite às aplicações processarem várias ações paralelamente. Por exemplo, enquanto um *thread* lê blocos de um arquivo em disco, outro *thread* pode paralelamente processar as entradas do usuário. Esse serviço permite um melhor aproveitamento de *hardware* multiprocessador, sendo ideal para a programação de aplicações com paralelismo inerente [8].

2.1.3 Chamada de Procedimento Remoto e Serviço de Apresentação

O serviço de Chamada de Procedimento Remoto, ou **RPC**[3], fornece um paradigma de alto nível que permite a comunicação entre processos que residem em diferentes espaços

---

[3] *Remote Procedure Call*

de endereçamento de uma ou mais máquinas. Com o serviço de RPC, o programador pode desenvolver uma aplicação que chame uma função (procedimento remoto) que será executada por outro processo de outra máquina. Do ponto de vista do programador, chamar um procedimento remoto é análogo a chamar um procedimento local.

Em um ACD, é possível que os parâmetros e resultados das RPCs, feitas entre máquinas com arquitetura diferentes, tenham representação interna de dados incompatíveis, causando problemas. O serviço de RPC usa o **serviço de apresentação** para normalizar as diferentes representações de dados, possibilitando a chamada de procedimentos remotos em ambientes heterogêneos.

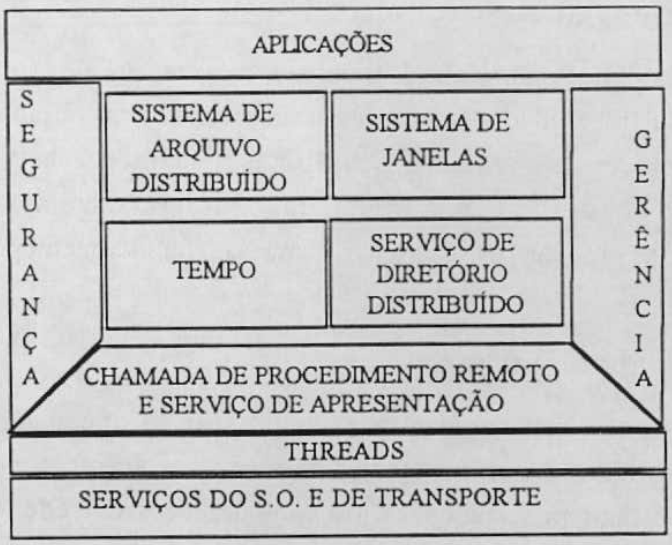

Figura 1. Arquitetura e Serviços Típicos de um Ambiente de Computação Distribuída

2.1.4 Serviço de Diretório Distribuído

Em um ACD, é comum a necessidade de localizar um determinado recurso que esteja disponível na rede. O serviço de diretório distribuído (***naming***) fornece um modelo de nomes únicos que permite a localização de qualquer recurso no ACD, independentemente da sua localização física na rede [9] [10] [11].

2.1.5 Serviço Distribuído de Tempo

Muitas aplicações necessitam de uma referência de tempo para escalonar suas atividades e determinar a sequência e duração de eventos. Em um ACD, a existência de várias máquinas leva à existência de múltiplas referências de tempo. Portanto, os diferentes componentes do ACD podem obter tempos diversificados nos computadores da rede. O objetivo do serviço de tempo é sincronizar os relógios existentes nos vários computadores, fornecendo um **tempo padrão** único e confiável para todas as aplicações executadas no ACD. Os *logs* de bancos de dados são um exemplo de aplicação desse serviço [12].

2.1.6 Sistema de Arquivos Distribuído

Em ACDs, é comum a necessidade de compartilhar informações que estão armazenadas em arquivos distribuídos pela rede. O sistema de arquivos distribuído, ou DFS[4], atende a essa necessidade integrando os arquivos existentes na rede, formando um sistema de arquivos global. Essa integração resulta em uma mesma semântica, tanto para arquivos locais quanto para arquivos remotos. O usuário pode acessar os sistemas de arquivos remotos como se estes fossem locais, independentemente de sua localização física na rede. O DFS está integrado com outros serviços, como por exemplo, RPC, *naming* e tempo [13] [14].

2.1.7 Sistema de Janelas

O sistema de janelas (*Network Windowing System*) é um serviço que permite que a tela da estação de trabalho do usuário seja subdividida em várias partes denominadas de janelas. Cada janela representa uma aplicação que pode estar sendo executada em qualquer máquina da rede. Isso torna possível, para o usuário, acompanhar os resultados da execução de várias aplicações, locais e remotas, simultaneamente nas janelas de uma mesma tela [15] [16].

2.1.8 Serviços de Segurança

Na maioria dos sistemas centralizados, o sistema operacional é o responsável pela verificação da identidade do usuário e pela permissão de acesso aos recursos. Em um ACD, essas atividades estendem-se pelos vários computadores da rede, requerendo serviços de segurança independentes e confiáveis. O principal objetivo dos serviços de segurança é proteger os dados e recursos do ACD contra acesso, modificação ou destruição por usuários não autorizados.

2.1.9 Serviço de Gerência Distribuída

Em um ACD, a existência de diversos ambientes computacionais, com diferentes esquemas de administração, gera a necessidade da utilização de um serviço de gerência global que integre os esquemas de administração existentes. O serviço de gerência distribuída fornece meios para uma administração eficiente e uniforme dos sistemas, das redes e das aplicações do usuário integradas ao ACD [4] [17] [18].

Os serviços de ACDs oferecem alternativas para parâmetros e outras características operacionais, que devem ser selecionadas criteriosamente pelos programadores, de forma a atender requisitos funcionais e/ou de execução (como nível de desempenho, confiabilidade, segurança, facilidade de uso, etc.) das ADs que eles projetam e codificam. O objetivo do artigo é apresentar diretrizes para a seleção criteriosa das alternativas possíveis. Para tanto, é necessário primeiro, que se identifiquem os requisitos (genéricos) de ADs.

---

[4] DFS significa *Distributed File System*. Alguns autores adotam o termo *Distributed File Service*.

3 Requisitos de Aplicações Distribuídas

Ao projetar uma AD, o programador deve considerar um conjunto básico de requisitos que podem ter impacto direto na qualidade da implementação. Uma meta que todo projeto de AD deve procurar atingir é transparência. Outras metas importantes são consistência e eficiência, obtidas a partir das considerações de heterogeneidade, confiabilidade, escalabilidade e desempenho.

3.1 Transparência

Todo projeto de AD deve procurar tornar a existência de computadores, geograficamente distribuídos pela rede, transparente para os usuários. Isso permite acessar os recursos da rede, usando os mesmos nomes e operações, independentemente de suas localizações. Para isso, o projeto da AD deve considerar as seguintes possibilidades [19]:

• **Transparência de Acesso**: permite que o acesso aos recursos, como por exemplo, arquivos, impressoras, CPUs, banco de dados, processos locais e remotos, seja feito através do uso de um mesmo conjunto de operações.

• **Transparência de Localização**: permite que os recursos sejam acessados independentemente de sua localização real. Por exemplo, nomes de arquivos dependentes do nome da máquina como *máquina1.notas.txt* devem ser evitados.

• **Transparência de Concorrência**: habilita vários usuários a realizarem operações concorrentemente sobre os dados e recursos compartilhados, sem que haja efeitos de interferência de um sobre o outro.

• **Transparência de Replicação**: admite que múltiplas cópias de arquivos e outros recursos sejam usados para aumentar confiabilidade ou desempenho, sem que os usuários tenham conhecimento da existência de réplicas.

• **Transparência de Falhas**: permite que a ocorrência de falhas de *hardware* ou *software* sejam "escondidas" do usuário, de tal forma que os usuários possam completar suas tarefas independentemente das falhas.

• **Transparência de Migração**: permite a movimentação de recursos entre os vários sistemas sem que as operações do usuários sejam afetadas.

• **Transparência de Escala**: permite que a aplicação seja expandida em escala sem que haja a necessidade de mudanças na estrutura do sistema ou nos algoritmos da aplicação.

3.2 Heterogeneidade

Os módulos da AD devem ser projetados para tratarem possíveis problemas causados pelas diferenças entre linguagens de programação, sistemas operacionais, representação de dados, esquemas de segurança e protocolos de comunicação. Se isso não for considerado, a AD pode apresentar problemas quando seus processos residirem em plataformas diferentes. O tratamento dos efeitos de heterogeneidade requer o uso do serviço de apresentação do ACD.

3.3 Confiabilidade

Como visto na Seção 1, uma das principais vantagens das ADs é a possibilidade de se obter aplicações confiáveis, independentemente da ocorrência de falhas. Para isso, devem ser considerarados três requisitos: disponibilidade, tolerância a falhas e segurança.

3.3.1 Disponibilidade

A disponibilidade pode ser definida como a quantidade de tempo contínuo que a AD é utilizável. Um meio de se aumentar a disponibilidade é através da **replicação** dos módulos vitais da AD (*hardware* e *software*). Se qualquer recurso usado pela AD falhar, as réplicas podem ser usadas para substituí-lo, permitindo que o usuário continue executando normalmente suas tarefas.

Para se obter boa disponibilidade, não basta apenas ter as réplicas disponíveis, é necessário que elas estejam íntegras quando requisitadas. Portanto, o programador deve tomar os cuidados necessários, para manter as réplicas atualizadas, a fim de evitar problemas de inconsistência.

3.3.2 Tolerância a Falhas

Dotar a AD de alta disponibilidade não é condição suficiente para que ela tenha alta confiabilidade. São necessários cuidados adicionais para garantir que, em caso de falhas, não exista o comprometimento da integridade das operações já realizadas pela AD e nem riscos para as operações futuras. Uma AD que tem a propriedade de continuar funcionando corretamente, após a ocorrência de falhas, é considerada uma aplicação tolerante a falhas. Quando se deve ou não dotar a AD dessa propriedade, depende essencialmente dos requisitos do problema que está sendo modelado. Um exemplo prático de tolerância a falhas é a solução adotada pelo protocolo do *Network File System* (NFS) da Sun Microsystems [11]. O NFS evita inconsistências, após a ocorrência de falhas, simplesmente não mantendo nenhum tipo de estado. A ausência de estado permite que as operações sejam reiniciadas após uma falha, como se nada tivesse ocorrido.

3.3.3 Segurança

Projetar uma AD segura requer cuidados especiais principalmente com as operações de comunicação entre os processos remotos. Essa comunicação geralmente é feita através da troca de mensagens transmitidas através da rede. Os processo da AD não pode simplesmente assumir que as mensagens que recebem têm origem confiável. Existe a possibilidade das mensagens serem capturadas e falsificadas, por usuários mal intencionados que estejam monitorando a rede. O projeto da AD deve considerar a possibilidade de uso dos seguintes serviços de segurança, normalmente disponíveis nos ACDs: canal seguro de comunicação, identificação, autenticação, autorização e auditoria.

• **Canal Seguro de Comunicação**: garante a privacidade e integridade dos dados transmitidos pela rede. A **privacidade** faz com que os dados não sejam visualizados, quando estiverem sendo transmitidos através da rede. A **integridade** deve garantir que os

dados não sejam alterados ou corrompidos enquanto estiverem sendo transmitidos através da rede.

• **Identificação:** é o processo pelo qual uma entidade (usuário ou recurso) reclama uma identidade indiscutível previamente estabelecida. O serviço de identificação tem a responsabilidade de definir identificadores globais únicos (***netnames***), para tornar possível a implementação de outros serviços de segurança, tais como, autenticação, autorização e auditoria [9] [5].

• **Autenticação**: é o processo pelo qual uma entidade tem a sua identidade verificada, para eliminar possíveis dúvidas quanto à questão dela ser realmente quem diz ser. Toda entidade cuja identidade pode ser verificada pelo serviço de autenticação é tida como um *principal*. O termo *principal* está associado a máquinas, usuários, processos, serviços e dispositivos confiáveis. Os serviços de autenticação fazem uso de protocolos baseados em criptosistemas simétricos e/ou de chave pública.

• **Autorização**: determina quem pode acessar que recurso do ACD e como. O esquema de autorização é controlado através da associação de permissões aos recursos de um determinado domínio da rede. Listas de controle de Acesso (ACLs) normalmente são usadas para esse fim [19].

• **Auditoria**: tem como objetivo permitir o monitoramento das ações dos usuários, para identificar possíveis problemas de segurança (quem fez o que e quando). Esse serviço permite detectar comportamentos impróprios, mas somente após a sua ocorrência. As informações das operações sobre dados, recursos, sucesso e insucesso de *logins*, comandos executados por cada usuário, chamadas remotas e autenticações são armazenadas em um banco de dados, para, posteriormente, serem analisadas pelo administrador da rede.

3.4 Escalabilidade

Uma AD deve ser projetada para funcionar corretamente com desempenho aceitável tanto em um ACD com duas máquinas quanto em um ACD com milhares de máquinas. Essa característica é conhecida como escalabilidade [15]. Um projeto escalável deve gerenciar os picos de carga de forma elegante, adaptando-se ao crescimento do número de usuários e permitindo fácil integração com novos recursos adicionados ao ACD.

3.5 Desempenho

Uma AD pode atender aos requisitos de transparência, confiabilidade, escalabilidade, mas se o desempenho for crítico, a sua utilização pode se tornar inviável. O maior problema a ser contornado pelo programador é a relação existente entre o desempenho da AD e o desempenho da infra-estrutura de comunicação. Geralmente, a origem do problema está na troca de mensagens através da rede, necessária para a comunicação entre os processos remotos.

Considerações práticas sobre os principais requisitos de ADs são discutidos na próxima seção.

4 Considerações Específicas de Projeto

Nesta seção, serão apresentadas considerações específicas para o projeto de ADs, envolvendo o uso de sistemas de arquivos distribuído, seleção de serviço transporte, tratamento e recuperação de erros e desempenho. As considerações desta seção serão baseadas no serviço de Chamada de Procedimento Remoto[5] [20], por ser, atualmente, o mais usado para o desenvolvimento de ADs. Apesar disso, as considerações apresentadas também podem ser úteis para outros modelos de programação.

4.1 Uso de Sistema de Arquivo Distribuído

É comum em projetos de ADs, processos necessitarem atualizar ou compartilhar arquivos que se encontrem em diferentes máquinas da rede. Uma alternativa para solucionar esse problema, é usar um serviço de arquivos distribuído (DFS) para realizar as operações remotas que envolvem arquivos. Essa abordagem simplifica a implementação de ADs porque todo o tratamento de operações com arquivos remotos é feito pelo DFS e não pela aplicação, minimizando o trabalho de programação. Entretanto, o programador precisa certificar-se de que os requisitos da AD a ser desenvolvida não ficam comprometidos por restrições que alguns DFS impõem.

Projetos que utilizam arquivos montados[6] com DFS devem prever as implicações que o modelo semântico de compartilhamento de arquivos pode ter no funcionamento da aplicação. O modelo semântico de compartilhamento define precisamente os efeitos causados pelas operações de leitura e gravação, quando dois ou mais usuários compartilham um mesmo arquivo.

4.1.1 Implicações da Semântica de Compartilhamento

Um exemplo prático de implicações da semântica de compartilhamento pode ser observado na Figura 2, que usa o *Network File System* (NFS) da Sun. Inicialmente, o usuário faz uma consulta e, em seguida, imprime um arquivo que contém dados compartilhados de estoque. Entre o momento que o usuário faz a consulta e o momento da impressão, um outro usuário atualiza o arquivo de estoque no servidor de arquivos. Por causa da semântica de compartilhamento do NFS, o resultado da impressão conterá os dados desatualizados.

Esse problema deve-se ao fato do NFS não possuir uma semântica de compartilhamento de arquivos bem definida. Essa indefinição é consequência de um esquema de *cache* usado pelo NFS para otimizar desempenho [13][11]. Isso significa que, quando um arquivo montado com NFS é compartilhado por clientes e servidores de diferentes máquinas, existe a possibilidade das operações de escrita, realizadas por um

---

5 O serviço de RPC é normalmente baseado no modelo cliente-servidor. O processo que chama um procedimento remoto é denominado de cliente. O processo que executa o procedimento remoto e, possivelmente, retorna um resultado para o cliente, é denominado de servidor.

6 "montar" um arquivo significa coloca-lo disponível em uma determinada máquina. Isso é normalmente feito através de um comando denominado de *mount*.

determinado processo, não serem imediatamente visíveis pelos outros processos que compartilham o arquivo. Somente após o NFS atualizar a *cache* da máquina A, é que a impressão seria processada corretamente.

Figura 2. Problemas causados pela semântica de compartilhamento de arquivos do NFS

O desconhecimento dos problemas causados pelos efeitos da semântica de compartilhamento do NFS podem induzir o programador a procurar erros inexistentes no código-fonte da aplicação. No caso do exemplo apresentado, o programador poderia concluir erroneamente que o arquivo de estoque não estava sendo atualizado corretamente pelo usuário 2 ou que a aplicação do usuário 1 não os estaria lendo corretamente do arquivo. A procura equivocada do "erro", no código-fonte das aplicações, resultaria em perda de tempo por parte do programador, sem resolver o problema.

O fato do NFS não possuir uma semântica de compartilhamento bem definida, transfere para o programador a responsabilidade de tratar os problemas, a nível de aplicação. Através do uso de alguns *system calls* é possível forçar a atualização das *caches* do NFS, adequando sua semântica de compartilhamento, aos requisitos da aplicação. Obviamente, essa solução pode penalizar desempenho.

Nem todos os DFS apresentam esse problema. Se no lugar do NFS estivesse sendo utilizado o *Andrews File System* (AFS) da Transarc [21], o resultado da impressão seria correto. O AFS provê semântica de compartilhamento bem definida, que garante a disponibilidade dos dados atualizados pelos usuário, indepedentemente da máquina em que se encontrem.

4.2 Seleção de Protocolos de Transporte

Um aspecto importante a ser considerado no projeto de ADs é a definição do protocolo de transporte que viabiliza a comunicação entre os processos remotos da AD. Os serviços de RPC oferecem pelo menos dois tipos de transporte: orientado a conexão e sem conexão. Um protocolo orientado a conexão, bastante usado atualmente, é o *Transmission Control Protocol* (TCP). Já o *User Datagram Protocol* (UDP) é um exemplo de protocolo sem conexão [22]. Analisando as vantagens e desvantagens de TCP e UDP, o programador tem subsídios para identificar o transporte mais adequado aos requisitos da AD ser desenvolvida.

Os principais critérios que devem ser considerados na seleção do protocolo de transporte são: confiabilidade, desempenho, semântica de chamada remota, idempotência, capacidade de dados e escalabilidade.

4.2.1 Confiabilidade

O uso de TCP permite a troca confiável de dados entre clientes e servidores. A existência de uma conexão garante que nenhum pedido de RPC é perdido ou recebido fora de ordem pelo servidor. Com UDP o cliente não tem garantias de que o servidor recebe os pedidos enviados. Por ser um transporte não confiável, pode haver, além da perda, replicação ou troca de ordem dos pedidos. Em função disso, os usuários de UDP devem considerar a possibilidade do tratamento de erros de pedidos de RPC a nível de aplicação. Existem casos particulares em que esse tratamento pode ser desnecessário. Por exemplo, se a AD vai ser executada somente em uma rede local tipo *Ethernet*, o *hardware* oferece garantias de que não haverá perdas de datagramas UDP.

4.2.2 Desempenho

A maior vantagem de UDP sobre TCP é o desempenho. Uma conexão TCP consome tempo e requer manutenção de estado. A existência de estado implica no consumo de recursos do sistema, introduzindo *overhead*. Já UDP é um transporte mais leve que consome recursos somente quando os dados são enviados ou recebidos.

Apesar de UDP apresentar melhor desempenho, há casos em que uma conexão TCP pode ser mais eficiente. Por exemplo, quando a frequência (e o volume) de troca de dados entre o cliente e servidor for extremamente alta. Se a frequência for baixa, então UDP provavelmente é a escolha mais eficiente.

Uma estratégia simples para avaliar o impacto do protocolo de transporte sobre o desempenho da AD é escrever pequenos programas de teste com as principais RPCs executadas pela AD. Os programas de teste devem utilizar cada uma das opções de transporte disponíveis para avaliar, individualmente, o desempenho de cada transporte. Os dados de desempenho dos programas de testes podem ser obtidos com ferramentas que traçam o perfil de execução dos módulos da aplicação. Os utilitários como *time* e *prof* (UNIX), são exemplos de ferramentas que podem ser usadas para esse fim.

4.2.3 Semântica de Chamada Remota

Quando um procedimento local é chamado e um resultado é retornado para quem fez a chamada, não existem dúvidas de que ele foi executado exatamente uma vez. Entretanto, no caso das RPCs, se o cliente recebe o resultado do servidor, isso não significa necessariamente que o procedimento remoto tenha sido executado exatamente uma vez. Isso normalmente ocorre quando as RPCs utilizam protoclos de transporte sem conexão. Em função disso, existem várias semânticas possíveis para as RPCs; as principais são: exatamente-uma-vez, no-máximo-uma-vez e pelo-menos-uma-vez.

Exatamente-uma-vez significa que o procedimento remoto é executado uma única vez. Esse tipo de operação é difícil de se obter, porque existe a possibilidade de falha do servidor. **No-máximo-uma-vez** significa que o procedimento remoto ou é executado exatamente uma vez ou não é executado nenhuma vez. Quando o cliente executa a RPC com sucesso, o procedimento remoto é executado uma vez. Por outro lado, se por algum motivo a RPC falhar, o cliente não sabe ao certo se o procedimento remoto foi ou não executado. **Pelo-menos-uma-vez**: significa que o procedimento remoto é executado, no mínimo, uma vez. Assumindo que o cliente retransmite um mesmo pedido até receber uma resposta válida do servidor, então existe a possibilidade do procedimento remoto ser executado mais de uma vez.

A semântica da RPC está diretamente relacionada com o tipo de protocolo de transporte usado pela RPC. O uso de transportes não confiáveis, como UDP, impõem uma semântica de chamada de pelo-menos-uma-vez. O UDP não garante que os pedidos das RPCs não sejam duplicados, podendo levar o procedimento remoto a ser executado mais de uma vez.

Com TCP, a RPC tem semântica de no-máximo-uma-vez. A conexão TCP garante que, se uma resposta for recebida pelo cliente, o procedimento remoto foi executado exatamente-uma-vez. Infelizmente, a semântica de exatamente-uma-vez nem sempre é verdadeira com TCP. No caso de falha do servidor, o cliente pode não receber nenhuma resposta e, portanto, não tem como saber se o procedimento remoto foi executado zero ou uma vez.

4.2.4 Idempotência

Os procedimentos que podem ser executados mais de uma vez sem causar transição de estado, no servidor, ou mudança do resultado para o cliente, são tidos como idempotentes. Por exemplo, a soma de dois números é um procedimento **idempotente**, porque, independentemente do número de vezes que a soma seja executada, o resultado é sempre o mesmo. Os procedimentos que não apresentam essa característica são considerados não idempotentes. Por exemplo, um procedimento para excluir um arquivo é **não idempotente**. Se após a exclusão do arquivo, realizada pela primeira execução, o procedimento de exclusão for novamente executado, ocorrerá um erro, pois o arquivo já não existe mais.

Quando os procedimentos da AD são idempotentes, o uso de UDP não tem contra indicação. O uso de UDP é desaconselhável nos casos em que os procedimentos são não idempotentes. O motivo é simples: como visto na seção 1.3.3, a semântica das RPCs, baseadas em UDP, é de pelo-menos-uma-vez. Isso significa que os procedimentos podem ser executados mais de uma vez, podendo assim haver violação da especificação da AD. Os procedimentos não idempotentes exigem semântica de no-máximo-uma-vez, que pode ser obtida com TCP.

4.2.5 Capacidade de Dados

O TCP leva vantagem sobre UDP quando muitos parâmetros e resultados têm que ser enviados ou recebidos pelas RPCs. Muitas implementações de UDP limitam o tamanho máximo do datagrama em 8k *bytes*. É possível contornar essa limitação, dividindo os dados em partes menores que 8k *bytes* e fazendo chamadas RPCs com cada uma das partes. Essa pode não ser uma boa solução, pois o aumento do número de RPCs pode penalizar o desempenho. Além disso, o programador tem que escrever código adicional para tratar o particionamento dos dados. Normalmente, RPCs que necessitam de parâmetros ou resultados muito grandes devem fazer uso de TCP, que não apresenta a limitação de tamanho imposta pelo datagrama UDP.

4.2.6 Escalabilidade

O uso de TCP implica sempre na abertura de uma conexão entre o cliente e o servidor, o que consome recursos. Em função disso, o servidor tem o número de conexões limitado pelos recursos do sistema operacional da máquina onde ele é executado. Se um número muito grande de clientes acessarem simultaneamente o servidor, então existe a possibilidade dos recursos gerenciados pelo sistema operacional serem insuficientes para atender a demanda das conexões, causando problemas. O UDP só envia datagramas, logo, teoricamente, não impõe limitações para que o servidor seja acessado por um número ilimitado de clientes.

4.3 Tratamento e Recuperação de Erros

Considera-se agora, a prevenção e o tratamento e recuperação de erros em ADs. Serão abordados os principais tipos de erros, suas causas e implicações. Também serão discutidas algumas soluções gerais para os problemas abordados, visto que soluções específicas, normalmente, são dependentes dos requisitos da AD.

4.3.1 Estratégias para Tratamento e Recuperação de Erros

Ao desenvolver uma AD, uma questão a ser resolvida pelo programador é o tratamento do término anormal de RPCs. Um término anormal ocorre, quando o cliente faz uma RPC e não recebe a resposta do servidor. Para evitar que o cliente fique aguardando indefinidamente uma resposta do servidor, os mecanismos RPCs geralmente adotam uma técnica de *timeout* ou equivalente. Se após a RPC, transcorrer um determinado tempo e o cliente não receber a resposta do servidor, então o serviço de RPC assume que houve um término anormal e retorna um código de *timeout* para o cliente.

Um problema enfrentado pelo programador é como definir que ação deve ser tomada, quando o cliente recebe um código de *timeout* sujeito a múltiplas interpretações: o servidor falhou? a rede está congestionada?. De forma simplificada, as estratégias adotadas pelo cliente, após detectar um *timeout*, podem ser: somente detectar o erro, procurar tratar o erro e recuperar o erro.

• **Somente Detectar o Erro**: ao receber uma indicação de *timeout*, o cliente simplesmente emite uma mensagem de erro para o usuário do tipo "Erro de *timeout*: servidor não responde". O usuário é responsável por tentar executar novamente a operação que falhou.

• **Procurar Tratar o Erro**: após detectar o primeiro *timeout*, o cliente repete a RPC algumas vezes para tentar obter sucesso na sua execução. Se, após as repetições, persistir o *timeout*, o cliente assume que o serviço não está disponível e emite uma mensagem de erro para o usuário.

• **Recuperar o Erro**: ao receber um (ou mais) *timeout* e assumir a indisponibilidade do serviço, o cliente pode adotar uma das seguintes estratégias para recuperar o erro: a) reinicializar o servidor e restaurar o seu estado, caso ele mantenha algum; b) executar o serviço no servidor de outra máquina, caso exista uma réplica.

As estratégias sugeridas acima não descrevem vários detalhes que, na prática, devem ser considerados pelo programador. Para determinar a melhor estratégia de tratamento e recuperação de erros, é fundamental que o programador tenha em mente os principais tipos de falhas que podem ocorrer, bem como suas causas e consequências. A análise dessas possibilidades vai permitir a elaboração de medidas preventivas que podem, inclusive, minimizar a ocorrência de *timeout*.

Assumindo que o código do cliente e do servidor estão livres de erros, a maioria dos erros em ADs estão relacionados com: o tipo do transporte usado pelas RPCs, problemas de desempenho, valor de *timeout* subdimensionado, *deadlock*, término anormal do servidor e término anormal do cliente. Cada uma dessas possibilidades tem implicações que exigem soluções específicas e que serão discutidas nas sub-seções a seguir.

4.3.2 Retransmissão Automática do Serviço de Transporte

Para minimizar parte do esforço de programação com o tratamento de perda de pedidos e respostas, muitos mecanismos RPC, quando usam um transporte não confiável, reenviam automaticamente o pedido enquanto não recebem a resposta do servidor ou atigem a condição de *timeout*.

As retransmissões podem causar múltiplas execuções do procedimento remoto mesmo sem que o cliente receba a indicação de *timeout*.. Outra possibilidade é o cliente receber o *timeout* somente após o procedimento remoto ter sido executado várias vezes pelos pedidos retransmitidos. Quando o procedimento remoto é idempotente, a ocorrência de execuções indesejáveis não compromete a integridade da AD. Mas, podem existir implicações, se o processamento do procedimento idempotente for longo e consumir muitos recursos. Então, nesse caso, múltiplas execuções desnecessárias podem criar problemas de desempenho. O problema mais grave a ser resolvido pelo programador são as execuções indesejáveis de procedimentos remotos não indempotentes. Uma solução para tratar esse problema será discutida a seguir.

4.3.2.1 Múltiplas Execuções e Idempotência

O programador pode evitar execuções indesejáveis de procedimentos remotos a nível de aplicação, adotando um esquema de **cache de resposta** no servidor. Nesse esquema, cada RPC é acrescida de um parâmetro representando um identificador único. Esse identificador é usado para evitar que as RPCs sejam executadas mais de uma vez ou que haja troca na ordem de execução. A implementação pode ser feita da seguinte forma: no cliente, acrecenta-se o identificador do pedido de serviço a ser submetido ao servidor. No servidor, o programador deve criar a *cache* de resposta para armazenar informações sobre todos os procedimento remotos executados. Todos os procedimentos remotos que necessitarem tratar execuções indesejáveis devem ser escritos de tal forma que sempre consultem a *cache* de resposta, antes de executarem o serviço requerido pelo cliente [23].

4.3.3 Problemas Relacionados com Desempenho

Alguns erros de *timeout* são causados por problemas de desempenho, tais como: congestionamento na rede, sobrecarga na máquina onde o servidor é executado ou tipo inadequado de servidor. O congestionamento da rede pode causar o atraso excessivo dos pedidos e respostas das RPCs, levando o cliente a receber um *timeout*. Quando a máquina do servidor está sobrecarregada, a execução do procedimento remoto pode ficar lenta, a ponto do cliente receber um *timeout* antes do término da execução. Finalmente, a última consideração é quanto ao tipo do servidor. Por exemplo, se o servidor é iterativo e muitos clientes enviam simultaneamente vários pedidos para ele, então pode haver demora no atendimento dos pedidos enfileirados, retardando as respostas e causando um *timeout*.

4.3.4 Erros Causados por Timeout Subdimensionado

Em geral, os mecanismos RPCs permitem que o programador especifique o intervalo de tempo do *timeout*. Se o programador definir um valor de *timeout* muito pequeno, o cliente pode receber uma indicação prematura de erro antes do servidor concluir a execução do procedimento remoto. Para definir corretamente o valor do *timeout*, o programador precisa considerar algumas particularidades do cliente e do servidor. Os servidores provavelmente processam os serviços em intervalos de tempo variáveis, dependendo de fatores como a carga do servidor, roteamento ou congestionamento da rede. O cliente deve ser projetado, para assimilar a variação de tempo dos serviços, a fim de não congestionar a rede reenviando continuamente pedidos cujas respostas nunca chegam. Preferivelmente, o cliente deve adotar alguma estratégia tipo ***back off*** que consiste em ir aumentando exponencialmente o seu valor de *timeout*, à medida que as respostas do servidor não chegam [23]. Essa estratégia representa um esquema de *timeout* adaptativo e pode ser implementada multiplicando-se o valor do *timeout* inicial por 2. Se o cliente receber novamente um *timeout*, um novo valor é recalculado multiplicando-se o valor do último *timeout* por 4 e assim sucessivamente [22].

4.3.5 Deadlock Distribuído

Um conjunto de processos da AD está em *deadlock*, quando cada processo do conjunto está aguardando por um evento que somente outro processo, pertencente ao mesmo conjunto, pode causar. Como todos os processos do conjunto estão aguardando, nenhum deles pode causar evento algum, consequentemente, todos os membros do conjunto ficam retidos indefinidamente. Algumas alternativas para o tratamento de *deadlock* são: ignorar, impedir, detectar e limitar.

• **Ignorar**: consiste em simplesmente em ignorar a probabilidade da ocorrência de *deadlock*. Nesse caso, não existe nenhum esquema para tratar a ocorrência de *deadlock* a nível de aplicação.

• **Detectar**: esse esquema analisa a ordem das RPCs para identificar ciclos, a fim de detectar a ocorrência do *deadlock* e eliminá-lo.

• **Impedir**: essa estratégia consiste em tornar a ocorrência do *deadlock* estruturalmente impossível. O *deadlock* pode ser prevenido fazendo-se as RPCs em uma ordem pré-definida. Uma forma de se fazer isso é numerando os recursos usados pela AD, e exigindo que os processos ganhem acesso a eles em uma ordem estritamente crescente. Nesse esquema, um processo nunca pode manter para si um recurso de ordem mais alta e pedir um recurso de ordem mais baixa, eliminado assim a possibilidade de ciclos.

• **Limitar**: consiste em impor um limite de tempo (*timeout*), para aguardar a liberação dos recursos ou resposta ao pedido. Se o total de tempo é excedido, então assume-se que existe uma condição de *deadlock*. Alguns problemas dessa técnica são: a) em sistemas sobrecarregados, o número de RPCs que retornam *timeout* tende a aumentar; b) RPCs que têm tempo de processamento muito longo podem ser penalizados antes do término da execução. Informações detalhadas sobre técnicas para tratamento de *deadlock* distribuído podem ser encontradas em [24] e [19].

4.3.6 Término Anormal do Servidor

Quando um servidor sofre um término anormal, é necessário recriá-lo a fim de tornar seus serviços novamente disponíveis para os seus clientes. Os pontos básicos relacionados com o término anormal do servidor, que devem ser considerados pelo programador, são: reinicialização, recuperação de estado e recuperação com uso de replicação.

• **Reinicialização do Servidor**: existem várias abordagens para resolver o problema de reinicialização de um servidor. Uma alternativa é a reinicialização ser de responsabilidade do cliente - ao acessar um servidor sem sucesso e assumir que ele falhou, o cliente pode recriá-lo na mesma máquina em que ocorreu a falha ou em outra, para torná-lo novamente disponível.

Existem situações em que a reinicialização do servidor não é feita pelo cliente. O programador pode fazer uso de um servidor dedicado, ou **servidor de *boot***, para recriar os servidores que falharam [25]. O servidor de *boot* possui um cadastro com a identificação de

todos os servidores que devem ser monitorados. Todo servidor cadastrado, tem o seu ciclo de vida checado periodicamente. Em determinados intervalos de tempo, o servidor de *boot* verifica se os servidores sob sua responsabilidade estão operacionais. Assim que detecta a indisponibilidade de algum servidor cadastrado, o servidor de *boot* providencia imediatamente sua reinicialização.

• **Recuperação de Estado**: após reinicializar um servidor que falhou é necessário restaurar o seu estado, caso ele mantenha algum. As ações a serem tomadas para restauração de servidores que mantêm estado são fundamentais em ADs com requisitos de tolerância a falhas. A recuperação só é possível, se toda informação de estado for previamente armazenada, durante o ciclo de vida do servidor, em um meio não volátil (ex.: disco) que esteja sempre disponível. Dessa forma, durante o processo de reinicialização, o servidor pode restaurar o seu estado, lendo a informação previamente salva, voltando a atender os clientes de forma íntegra.

• **Recuperação com Replicação**: uma estratégia para tratar falhas de servidores é usar replicar serviços para aumentar a disponibilidade. Ao detectar a indisponibilidade do servidor, o cliente simplesmente redireciona as RPCs para uma réplica do serviço. Para localizar a réplica requerida, o cliente pode fazer uso do serviço de diretório distribuído.

O uso de replicação requer cuidados adicionais com relação à manutenção de estado. Quando existe replicação de servidores com estado, é necessário manter o estado das réplicas atualizados de tal forma que, quando ocorrer uma falha no servidor principal, os clientes possam usar os servidores replicados sem problemas. Essa abordagem pode ser complexa de implementar, porque as réplicas, obrigatoriamente, devem ter seus estados atualizados a cada transição de estado do servidor principal. Uma forma de se fazer isso é usando um serviço de *broadcast*. Toda vez que o estado do servidor principal sofrer alguma mudança, é emitida uma mensagem de *broadcast* com as informações necessárias para as réplicas atualizarem os seus estados [26].

4.3.7 Término Anormal do Cliente

O término anormal do cliente tem pelo menos três aspectos que devem ser observados pelo programador: problemas com órfão, reinicialização do cliente e recuperação de estado (caso o cliente mantenha algum).

• **Problemas com Órfão**: uma questão importante a ser considerada no projeto de ADs é o que acontece quando o cliente faz uma RPC e sofre um término anormal, antes de receber o resultado do servidor. Supondo que o pedido de RPC foi recebido pelo servidor, o procedimento remoto é executado sem que ninguém mais esteja aguardando o seu resultado. Esse processamento desnecessário é conhecido como **órfão**, porque o cliente responsável pelo pedido, morre antes de receber a resposta do servidor. Os órfãos podem causar vários problemas, como por exemplo, o consumo desnecessário de ciclos de CPU, travamento indesejável de arquivos ou *deadlock*.

Em situações em que o mecanismo de RPC não oferece tratamento de órfão, o programador tem que resolver esse problema a nível de aplicação. Três alternativas simples

são [27]: exterminação, reencarnação e expiração.

Na **exterminação**, o cliente armazena em um *log* informações sobre as RPCs efetuadas. O *log* deve residir em um meio não volátil para sobreviver à falha. Ao ser reinicializado após a falha, o cliente acessa o *log*, para obter as informações que permitam localizar e eliminar explicitamente os órfãos. A desvantagem dessa solução é o custo de leitura e escrita em disco do *log* a cada RPC. Para manter o *log* consistente, o cliente deve, toda vez que receber uma resposta do servidor, excluir do *log* o registro correspondente ao procedimento remoto executado. Outro problema dessa solução é que, antes de ser exterminado, um órfão pode fazer novas RPCs, dando origem a novos órfãos impossíveis de serem localizados através do *log* do cliente.

A **reencarnação** consiste em dividir o tempo de processamento das RPCs em **épocas** numeradas sequencialmente. Ao ser reinicializado, o cliente emite uma mensagem de *broadcast* para todos os servidores, informando que sofreu uma falha e que vai ter início uma nova época. Ao receberem a declaração de início de época, os servidores tentam localizar os clientes responsáveis pelas RPCs em andamento. Se os clientes responsáveis não forem localizados, então o processamento associado a eles é considerado órfão e, em seguida, eliminados pelos servidores. Naturalmente, se uma parte da rede cair, alguns órfãos podem sobreviver. Entretanto, quando a rede retornar ao normal, suas respostas conterão números de épocas obsoletas, tornando fácil sua detecção e eliminação pelo cliente. Uma vantagem dessa abordagem é que não é necessário armazenar nenhuma informação em um meio não volátil.

A **expiração** consiste em determinar uma quantidade de **tempo padrão de execução** (TPE), para o servidor executar o procedimento remoto. Se o TPE for insuficiente para o término da execução, o servidor faz um pedido ao cliente de uma nova quota de tempo. Nesse contexto, se o cliente falhar e não for reinicializado antes do valor do TPE ser atingido, o servidor assume que está processando um órfão e o elimina. Um problema a ser resolvido é a escolha de um valor razoável para o TPE. Isso não é trivial, porque depende das características de processamento dos procedimentos remotos, que, naturalmente, têm requisitos diferentes.

A simples eliminação de órfãos não resolve todos os problemas. O programador deve considerar os efeitos colaterais que uma eliminação sem critério pode causar. Por exemplo, se um órfão que mantém um arquivo travado for morto sem nenhum critério, o travamento pode se manter e o arquivo ficar bloqueado indefinidamente. Maiores detalhes sobre tratamento de órfãos podem ser encontrados nas referências [28] e [29].

• **Reinicialização do Cliente**: um meio de reinicializar de forma transparente um cliente que falhou é usando o esquema de servidor de *boot* já descrito anteriormente. Muitos ACDs oferecem serviços para recuperação de processos que sofrem término anormal. Caso o serviço existente não atenda satisfatoriamente os requisitos da AD, então o programador deve implementar o seu próprio servidor de *boot*.

• **Recuperação de Estado**: a recuperação de estado do cliente pode ser feita de forma semelhante à recuperação de estado do servidor. A diferença é que, além do estado

poder ser salvo em um disco acessível pelo cliente, ele também pode ser salvo em um servidor. Se o estado está em um disco acessível pelo cliente, então a recuperação consiste em carregar essa informação de estado durante o processo de reincialização, para que o processamento possa ser retomado a partir do ponto em que foi interrompido. Por outro lado, se o estado for salvo no servidor, o cliente tem que, durante a sua reinicialização, fazer RPCs ao servidor a fim de obter as informações necessárias para a restauração do estado.

4.4 Considerações de Desempenho

Nesta seção, serão inicialmente sugeridas alternativas para maximizar o desempenho de ADs, tais como: *cache* no cliente e no servidor, paralelismo, replicação de serviços, manutenção de estado no cliente e avaliação da eficiência dos serviços de transporte. Finalmente, serão discutidos alguns fatores que podem contribuir para a degradação de desempenho das ADs, como por exemplo, o uso sem critérios de serviços de *broadcast* e volume de dados dos parâmetros das RPCs.

4.4.1 Uso de Cache no Cliente e no Servidor

Uma forma de maximizar o desempenho das ADs é reduzir o número de RPCs, efetuadas pelos processsos da aplicação. Isso é importante por dois motivos: a) reduz o tráfego de mensagens de RPCs na rede, diminuindo a possibilidade de congestionamentos; b) reduz a carga de trabalho dos servidores, tornando-os mais eficientes.

Para reduzir o número de RPCs, o programador pode usar *cache* de pedido no cliente e *cache* de resposta no servidor.

• **Cache de Pedido no Cliente**: a *cache* de pedido no cliente é usada para manter os resultados das RPCs mais recentemente realizadas pelo cliente. Antes de fazer qualquer RPC, o cliente deve consultar a sua *cache* de pedidos para tentar obter os resultados desejados, evitando RPCs desnecessária. O grande problema desse esquema é como manter as informações da *cache* atualizadas. Por exemplo, se um procedimento remoto lê dados de um arquivo compartilhado que é atualizado com frequência por múltiplos clientes, então existe a possibilidade de alguma informação armazenada na *cache* ficar obsoleta. Isso porque, nesse caso, a *cache* simplesmente mantém uma cópia local dos dados remotos. Para resolver problemas de manutenção da integridade da *cache* podem ser usados protocolos de coerência de cache [30].

• **Cache de Resposta no Servidor**: como descrito na seção 4.3.2.1, a *cache* de resposta no servidor evita a execução desnecessária de procedimentos que foram recentemente executados. Essa estratégia também pode ser usada para tornar os servidores mais eficientes. Através dessa técnica é possível minimizar o consumo desnecessário de recursos por parte do servidor, como por exemplo, de ciclos de CPU ou leitura em disco, reduzindo o tempo de retorno do resultado das RPCs para o cliente. A *cache* de resposta do servidor, adaptada para fins de melhoria de desempenho, tem os mesmos problemas de consistência da *cache* do cliente.

4.4.2 Paralelismo

Uma das principais vantagens oferecidas pelas ADs é a possibilidade de explorar paralelismo para obtenção de aplicações com alto grau de desempenho. O uso de RPCs combinadas com o serviço de *threads*, permite executar paralelamente procedimentos da AD em diferentes computadores do ACD, tornando algumas aplicações mais rápidas. O uso de paralelismo depende dos requisitos e características particulares de cada aplicação.

Para melhorar o desempenho com paralelismo, o programador pode adotar a seguinte estratégia: a) identificar e dividir os serviços da AD que podem ser executados em paralelo[7]; b) distribuir esses serviços pelas várias máquinas que compõem o ACD, para viabilizar o paralelismo; c) projetar o cliente, para chamar os serviços remotos da aplicação, usando o serviço de *threads*; d) projetar servidores concorrentes com o serviço de *threads*, para aumentar a eficiência de atendimento aos clientes.

O uso de paralelismo introduz alguns problemas. Muitos processos de uma AD, executados paralelamente em diferentes máquinas, podem ter necessidade de se comunicar com muita frequência. Isso pode aumentar o número de mensagens na rede, gerando problemas de desempenho. Portanto, é importante considerar e controlar o grau de **granularidade** do paralelismo no projeto da AD. A granularidade é definida como o tempo total de processamento entre as RPCs necessárias para a comunicação entre os processos da AD. Quanto menor for granularidade, mais tempo será gasto com comunicação via rede, e menos tempo com processamento de CPU.

4.4.3 Replicação de Serviços

Existem situações em que a demanda de determinados serviços tende a crescer em escala exponencial, causando problemas de desempenho. A replicação de servidores (e recursos) é um meio de aumentar a disponibilidade dos serviços e de distribuir a carga de trabalho da AD. As réplicas de servidores, em diferentes máquinas da rede, tornam possível a múltiplos clientes executarem determinados serviços paralelamente, resultando em maior eficiência. O problema de usar réplicas é que, em algumas situações, elas devem ser atualizadas (ex.: réplicas de servidores com estado), como já foi visto na seção 4.3.6.

4.4.4 Manutenção de Estado no Cliente

Os servidores são responsáveis pela maior parte do processamento realizado pelas ADs. Uma forma de melhorar o desempenho é reduzir a carga de trabalho dos servidores, para aumentar sua eficiência. Um caso particular é o dos servidores com estado. Supondo que n clientes solicitam simultaneamente serviços que requerem manutenção de estado, por parte do servidor, então o servidor tem que gerenciar e manter n informações de estado, o que, dependendo da taxa de crescimento do número de clientes, pode representar uma carga de trabalho considerável. Transferindo para cada cliente a responsabilidade pela manutenção das informações de estado, é possível aliviar parte da carga de trabalho do

[7] A unidade de paralelismo pode ser um processo ou uma instrução de programa. A abordagem adotada neste trabalho será em torno de processo, que intuitivamente é a mais natural.

servidor, tornando-o mais ágil.

4.4.5 Chamada de Procedimento Remoto em Lote

Existem situações em que o cliente necessita fazer várias RPCs, ou **grupo** de RPCs, mas não precisa obter uma resposta enquanto o servidor não receber e processar todo o grupo. Nesses casos, é possível aumentar a velocidade de processamento das RPCs como um todo. Para isso, o programador pode fazer uso de uma facilidade, normalmente oferecida pelo serviço de RPC, conhecida como **RPC *Batch***. O RPC *Batch* permite ao programador enfileirar no lado do cliente um grupo de RPCs, e então enviá-las, através da rede, em uma única mensagem para o servidor, reduzindo tráfego de mensagens na rede. Somente a última RPC do lote retorna um resultado para o cliente. Portanto, o uso de RPC *Batch* impõe o uso de um transporte confiável como TCP [1] [23].

4.4.6 Cuidados com o Uso de Broadcast

Muitas implementações de mecanismos RPCs oferecem ao programador a possibilidade de propagar uma simples RPC, através do serviço de *broadcast* da rede, para todos os servidores de um determinado domínio. Essa facilidade é conhecida como **RPC *Broadcast*** e normalmente usa um transporte não confiável tipo UDP.

O RPC *broadcast* deve ser utilizado de forma criteriosa, para não interferir no fluxo normal dos dados transmitidos através da rede, causando problemas de desempenho. O envio excessivo de mensagens de *broadcast*, prejudica toda a comunidade de usuários. Por exemplo, estações de trabalho sem disco são penalizadas pelo mal uso do *broadcast* porque seus dados, obrigatoriamente mantidos em discos remotos, compartilham e concorrem na rede com as mensagens de *broadcast*. Portanto, o programador deve sempre usar o RPC *broadcast* em conjunto com alguma política de moderação.

4.4.7 Volume de Parâmetros de Chamadas Remotas

Outra alternativa para maximizar o desempenho das ADs é reduzir o volume (tamanho e quantidade) dos parâmetros das RPCs. Em casos onde é inevitável o envio de um grande volume de dados, uma alternativa é usar técnicas de compressão de dados antes dos parâmetros serem enviados. Outra possibilidade é procurar fazer otimizações a nível do serviço de apresentação. Em alguns serviços de apresentação é possível declarar as estruturas de dados, de tal forma que o número de *bytes* necessários para representação dos parâmetros das RPCs, na forma canônica, sejam minimizados. Na referência [23], é descrito um exemplo em que uma simples mudança na forma de declarar uma determinada estrutura de dados reduziu, em 20 por cento, o número de *bytes* dos parâmetros a serem transmitidos através da rede.

5 Conclusões e Sugestões de Continuação do Trabalho

Após breve exame da arquitetura e serviços de um Ambiente de Computação Distribuída (ACD), o trabalho se concentrou em discutir os requisitos e necessidades de

Aplicações Distribuídas (ADs), para pautar a apresentação de considerações e recomendações sobre a seleção de alternativas funcionais de tais arquitetura e serviços. Uma seleção criteriosa por projetistas e programadores possivelmente melhora a qualidade (i.e., satisfação de usuários) de ADs.

As considerações e recomendações feitas consideraram, em particular, aspectos de sistemas de arquivos distribuído, da camada de transporte, tratamento de erros e desempenho.

A limitação da semântica de compartilhamento (i.e., visões desincronizadas do conteúdo de um arquivo por ADs distintas) do *Network File System* (NFS) da Sun, pode ser tratada com o uso de *system calls* para atualização das *caches* (ao preço de degradação de desempenho) ou evitada com o uso do *Andrews File System* (AFS) da Transarc.

O impacto das características de transporte no desempenho de ADs pode ser apreciado economicamente através de uma estratégia de desenvolvimento e execução de vários (pequenos) programas de teste. Recomendações sobre a adoção de TCP ou UDP foram feitas em termos de confiabilidade, desempenho e idempotência. Apesar das vantagens de desempenho, UDP não é recomendado para ADs com procedimentos não idempotentes.

As considerações sobre tratamento de erros abordaram principalmente aspectos relativos a temporização (*"timeout"*), *deadlock* e término anormal do código do servidor e do cliente.

As recomendações para melhorar desempenho consideraram o uso de *cache*, paralelismo de execução, composição de lotes de chamadas remotas e restrições quanto ao emprego de difusão (*broadcast*) de chamadas.

Espera-se que o conteúdo do artigo seja uma coletânea de pontos importantes a serem ponderados quando do projeto e codificação de ADs. Se assim for, então o artigo contribui oferecendo um guia prático e organizado de cuidados e recomendações que enriquecem as fases de projeto, codificação e testes da Engenharia de Software, no caso de ambientes distribuídos heterogêneos.

O presente trabalho tem continuidade natural através de "estudo-de-casos" onde se consideram classes de ADs para ilustrar as considerações apresentadas e as implicações resultantes das recomendações feitas. Os estudos-de-caso possibilitam ainda, a análise de custos (em termos de alongamento de prazos, linhas-de-código adicionais e esforço para montar e executar cenários de testes) para "distribuir" aplicações. Os resultados desta análise acrescentarão informações relevantes à gerência de desenvolvimento de *software* em projetos (cada vez mais populares) de "*down-*" ou "*upsizing*" e de distribuição da computação corporativa de forma geral, agora baseada na arquitetura cliente-servidor. Adianta-se que um destes estudos, o qual considera um serviço de *spooling* de impressão distribuído, está em andamento na UFPB e na Infocon Tecnologia.

Agradecimentos

Os autores agradecem o suporte financeiro do CNPq a este trabalho.

Referências Bibliográficas

[1] Bloomer, J. *Power Programming with RPC*, O'Reilly & Associates, Inc. 1991.

[2] Millikin, Michael D. *Distributed Computing Futures: DCE, ONC and Beyond*, Unix Expo, Gunstock Hill Associates, 31 Oct. 1991.

[3] Open Software Foundation. *Distributed Computing Environment, Overview*, Jan. 1992, pp. 1-12.

[4] Cypser, R. J. *Communications for Cooperating Systems - OSI, SNA, and TCP/IP*, Addison-Wesley, 1991.

[5] Commer, D. E. & Stevens D. L. *Internetworking with TCP/IP - Design, Implementation, and Internals*, volume II, Prentice Hall, 1991.

[6] Rose, M. T. *An Introduction to Management of TCP/IP - based internets*, Prentice Hall, 1991.

[7] Martin, James & Chapman, Kathleen K. & Leben, Joe. *System Application Architecture: Common Communications Suport for Distributed Applications*, Prentice-Hall, 1992.

[8] Powell, M. L. & Kleiman, S. R. & Barton, S. & Shah, D. & Stein, M. & Weeks, M. *SUN/OS Multi-Thread Achitecture*, USENIX, Winter'91, Dallas-Texas, pp. 65-79.

[9] Malamud, C. *Analyzing SUN Networks*, Van Nostrand Reeinhold. 1992.

[10] Open Software Foundation. *Guide to OSF/1: A Technical Synopsis*, O'Reilly & Associates, Inc. 1991.

[11] Stern, H. *Managing NFS and NIS*, O'Reilley & Associates, Inc. Jun., 1991.

[12] Fidge, Colin. *Logical Time in Distributed Computing Systems*, IEEE Computer, Aug. 1991, pp. 28-33.

[13] Levy, Eliezer & Silberschatz, Abraham. *Distributed File Systems: Concepts and Examples*, ACM Computing Surveys, Dec. 1990, Vol. 22, No. 4, pp. 322-374.

[14] Open Software Foundation. *File Systems in a Distributed Computing Environment*, A White Paper, Jul. 1991, pp. 1-7.

[15] Gray, Pamela. A. *Open Systems: A Business Strategy for the 1990s*, McGraw-Hill, 1991.

[16] Dunphy, Ed. *The UNIX Industry: Evolutions, Concepts, Architecture, Applications, and Standards*, QED Technical Publishing Group, 1991.

[17] Open Software Foundation. *Guide to OSF/1: A Technical Synopsis*, O'Reilly & Associates, Inc. 1991.

[18] Open Software Foundation. *Distributed Management Environment*, White Paper, 1991. pp. 1-8.

[19] Coulouris, George F. & Dollimore, Jean. *Distributed Systems: Concepts and Design*, Addison-Wesley. 1988.

[20] Birrell, Andrew D. & Nelson, Bruce Jay. *Implementing Remote Procedure Calls*, ACM Transaction on Computer Systems, Vol. 2, no. 1, Feb. 1984, pp. 39-59.

[21] Cohen, David L., *AFS : NFS on Steroids*, LAN Technology, março de 1993, pp-52-62.

[22] Stevens, W. R. *UNIX Network Programming*, Prentice Hall, 1990.

[23] Corbin, J. R. *The Art of Distributed Applications - Programming Techniques for Remote Procedure Calls*, Springer-Verlag, 1991.

[24] Spector, A. Z. *Distributed Transaction Processing Facilities*, In Mullender, S.(ed.): Distributed Systems, Reading (Mass.) Addison_Wesley, 1989, pp. 191-214.

[25] Bal, H. E. *Programming Distributed Systems*, Prentice-Hall, 1990.

[26] Joseph, T. A. & Birman K. P. *Reliable Broadcast Protocols* In: Mullender, S.(ed.): Distributed Systems, Reading (Mass.). Addison-Wesley, 1989. pp. 65-86.

[27] Tanenbaum, Andrew. S. *Modern Operating Systems*, Prentice Hall. 1992.

[28] Panzieri, Fabio & Shrivastava, Santosh K. *Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing*, IEEE Transactions on Software Engineering, vol. 14, no. 1, Jan. 1988, pp. 30-33.

[29] Ravindran, K. & Chanson, Samuel T. *Failure Transparency in Remote Procedure Calls*, IEEE Transactions on Computers, vol. 38, no. 8, Aug. 1989, pp. 1773-1187.

[30] Mullender, Sape. *Distributed Systems*, Addison-Wesley, 1989.