

PARTE 2 – COMUNICAÇÃO EM SISTEMAS DISTRIBUÍDOS

- 5. Redes para Sistemas Distribuídos
 - 5.1 Tipos
 - 5.2 Protocolos
- 6. Comunicação entre processos (IPC)
 - 6.1 Construções Básicas
 - 6.2 Operações básicas
 - 6.3 Comunicação Síncrona e Assíncrona
 - 6.4 Destino das Mensagens
 - 6.5 Comunicação Cliente/Servidor
 - 6.6 Comunicação em Grupo
- 7. Comunicação RPC – Remote Procedure Call
 - 7.1 Operação Básica
 - 7.2 Binding
 - 7.3 Semântica do RPC
 - 7.4 Implementação
- 8. Estudo de Casos
 - 8.1 UNIX sockets
 - 8.2 ISIS
 - 8.3 SunRPC

PARTE 2 – COMUNICAÇÃO EM SISTEMAS DISTRIBUÍDOS

Os componentes de um sistema distribuído são logicamente e fisicamente separados, eles precisam se comunicar para interagir. Se assumirmos que os componentes que fornecem e requerem acesso a recursos são implementados como processos, a comunicação entre um par de processos envolve operações tanto no processo que envia como no processo que recebe. Para que esta comunicação seja possível os processos comunicantes precisam: de regras como protocolos de comunicação, primitivas de comunicação e modelos de comunicação. Comunicação em sistemas distribuídos esta dividida em 3 capítulos iniciando com uma revisão de alguns aspectos de redes de computadores que são importantes no caso de sistemas distribuídos (capítulo 2). Os capítulos 3 e 4 tratam de modelos de comunicação entre processos em sistemas distribuídos, especialmente, o modelo cliente/servidor, comunicação em grupo e RPC. O último capítulo desta parte apresenta alguns exemplos de mecanismos de comunicação.

5. REDES E SISTEMAS DISTRIBUÍDOS

Este capítulo faz uma revisão nas redes de computadores no que diz respeito aos requisitos de comunicação para sistemas distribuídos. As facilidades de rede usadas em sistemas distribuídos são implementadas utilizando componentes de hardware (switches, interfaces) assim como componentes de software (protocolos e tratadores) e esta coleção de componentes é chamada de *subsistema de comunicação*. O projeto de um subsistema de comunicação é bastante influenciado pelas características do sistema operacional usado nos computadores que compõe o sistema distribuído assim como pela tecnologia de rede usada. Aqui vamos considerar alguns fatores que tem influencia no subsistema de comunicação.

Parâmetros de performance. Os parâmetros de performance de interesse são aqueles que afetam a velocidade de transferência das mensagens, latência e taxa de transferência.

- Latência pode ser definido como o tempo necessário para transferir uma mensagem vazia entre computadores. Significa os retardos de software para acessar a rede tanto no lado do enviante quanto no lado do recebedor, para obter acesso a rede e dentro da rede.
- Taxa de transferência é a velocidade de transferência dos dados entre computadores da rede, após iniciar a transmissão (bps).

Muitas mensagens em SDs são de tamanho pequeno e neste caso a latência é tão ou mais significativa que a taxa de transferência na determinação da performance.

A banda passante total do sistema é uma medida de desempenho, o volume total de tráfego que pode ser transferido pela rede em um dado instante. Em muitas LANs, incluindo Ethernet, a capacidade total de transmissão é usada para cada transmissão e a banda passante é igual a taxa de transferência. No caso das redes de longa distância, mensagens podem ser transferidas em vários canais diferentes de forma simultânea e a banda passante total não tem relacionamento direto com a taxa de transferência. A performance das redes se deteriora em condições de sobrecarga, ou seja, quando existe muitas mensagens ao mesmo tempo. O efeito da sobrecarga na latência, taxa de transferência e banda passante de uma rede depende muito da tecnologia usada.

Requisitos de performance. Se considerarmos uma comunicação do tipo cliente/servidor para acesso a arquivos compartilhados, deveríamos conseguir uma performance comparável ao acesso a arquivos em uma arquitetura centralizada onde a velocidade de acesso esta limitada pela performance do disco. Valores típicos de acesso a um bloco variam de 10 a 20 milisegundos. No caso cliente/servidor o tempo de acesso inclui o tempo de envio da requisição (dezenas de bytes) e a recepção da resposta (1K bytes). Este tempo não deveria ser maior do que o tempo de acesso ao disco (menor que 10 mili). Performance deste tipo é conseguida com latência menor que 5 mili e taxa de transferência maior que 200Kbytes por segundo, incluindo tempo de overhead de software.

Requisitos de confiabilidade. Garantias de confiabilidade são requisitados pela maioria das aplicações de sistemas distribuídos. A confiabilidade de comunicação é bastante alta na maioria dos meios de transmissão. Quando erros ocorrem são na maioria devidos a falhas

de temporização do software na transmissão e recepção do que na rede. A detecção de erros e sua correção é realizada pelo nível de aplicação.

5.1 Tipos de redes

De uma forma geral as redes de computador podem ser divididas em dois tipos : redes locais e redes de longa distância.

Redes Locais (LAN). Transferem mensagens a velocidades relativamente altas entre computadores conectados a um meio de comunicação tal como fibra ótica ou cabo coaxial no perímetro de um prédio ou campus. Não existe necessidade de roteamento das mensagens, visto que o meio prevê conexões diretas entre todos os computadores na rede. A latência baixa a não ser quando o tráfego é muito intenso. As taxas de transferência nas redes locais podem variar desde 0.2 até 100 Mbps sendo consideradas adequadas para implementação de sistemas distribuídos. Entretanto, algumas aplicações de multimídia necessitam taxas maiores.

Redes de longa distância (WAN). Transferem mensagens a velocidades baixas entre computadores separados por grandes distâncias (cidades, países, continentes). O meio de comunicação é um conjunto de circuitos de comunicação que ligam um conjunto de computadores dedicados chamados chaveadores de pacotes que gerenciam a rede. As mensagens seguem rotas para chegar ao destino e este roteamento implica em um retardo no tempo de transmissão. A performance atual destas redes não atende as necessidades de sistemas distribuídos. Valores típicos de latência variam entre 0.1 e 0.5 segundos sendo que a taxa de transmissão em redes de longa distância pode variar de 20 a 500 Kbps. Entretanto, com novas tecnologias como ISDN (Integrated Services Digital Network) e B-ISDN (Broadband-ISDN) espera-se um impacto maior no desenvolvimento destas redes. Em ISDN as taxas são múltiplos de 64Kbps (canal básico) e B-ISDN possibilita transmissões de 150Mbps ou mais.

5.2 Protocolos

O termo protocolo é usado para referir um conjunto de regras bem conhecidas e formatos para serem usados na comunicação entre processos. Duas partes são importantes na definição de protocolos: 1) especificação de uma sequência de mensagens que devem ser trocadas e 2) especificação de um formato de dados para as mensagens. O conjunto de protocolos que usados em um subsistema de comunicação é chamado de pilha de protocolos, onde os protocolos são arranjados em níveis. Exemplo de pilha de protocolo inclui o modelo de referência OSI (Open Systems Interconnection) da ISO (International Standards Organization), o padrão IEEE 802 para LANs, Internet e modelo cliente/servidor.

Em redes do tipo TCP/IP existem várias portas em cada computador com números bem conhecidos, cada uma destinada a um determinado serviço Internet tal como *telnet* ou *ftp* e para serviços UNIX do tipo *rlogin*. Para acessar um serviço em um dado *host*, uma requisição é enviada para a porta específica do serviço no *host*. Um sistema distribuído apresenta uma multiplicidade de serviços (servidores) que pode ser diferente de tempo em tempo de organização para organização. A alocação de *hosts* fixos ou portas fixas para

estes serviços não é possível. De uma forma geral dois tipos de serviço de transporte de dados estão disponíveis através dos protocolos: 1) orientado a conexão, onde uma conexão virtual é estabelecida entre o enviante e o receptor para a transmissão de uma sequência de dados (TCP) e, 2) sem conexão, onde mensagens individuais (datagramas) são transmitidos para destinos especificados (UDP). Em sistemas distribuídos as comunicações em geral seguem o modelo cliente/servidor que por sua vez normalmente usa transporte sem conexão para sua implementação. Isto principalmente devido ao overhead causado pelo estabelecimento de conexão. Outra característica em sistemas distribuídos é a necessidade de comunicação em grupo, considerando que um serviço pode ser provido por mais de um servidor. Outros itens importantes em protocolos para sistemas distribuídos incluem : mecanismos de criptografia flexíveis e eficientes objetivando a segurança, formas de tratamento automático (alocação de endereços) da parte de gerência da rede e, escalabilidade no que diz respeito a existência de um protocolo de comunicação apenas.

Protocolo FLIP (Fast Local Internet Protocol). Segundo Kaashoek [1993] existem vários requisitos que não são satisfeitos por protocolos do tipo TCP/IP e que deveriam ser levados em conta para sistemas distribuídos como transparência, segurança e gerência da rede. Para prover requisitos deste tipo foi proposto o protocolo FLIP, para ser usado num ambiente internetwork mais limitado formado por várias ethernet e computadores executando o sistema operacional distribuído Amoeba. O protocolo provê um serviço datagrama não confiável. Diferente de IP em FLIP os pontos fonte e destino dos pacotes são processo Amoeba (com identificadores de porta FLIP) e não computadores. Os identificadores de portas (64-bits) são gerados sempre que uma porta é criada e são endereços independente de localização. Os identificadores podem apontar grupos de portas. As mensagens em FLIP podem chegar a 4 Gbytes de tamanho. Cada computador em uma internetwork FLIP é conectado a rede física através de um 'FLIP box' que pode ser implementado em hardware ou software, contendo os seguintes componentes: 1) Chaveador de pacotes, passa pacotes entre hosts e redes e de uma rede para outra. Usa tabelas de roteamento (variação dinâmica) para escolher a rede onde os pacotes são transmitidos. 2) Interface do host, fornece interface aos protocolos FLIP para transmissão e recepção mensagens unicast, multicast e broadcast. Permite que processos registrem seus identificadores (criptografia função one way) para recepção de mensagens. 3) Interface de rede, um host com uma única interface e um roteador tem várias interfaces para diferentes redes.

A seguir as principais diferenças entre os protocolos FLIP e Internet são apresentadas:

- FLIP oferece transparência de localização, IP porta em computador
- Identificadores são gerados e alocados automaticamente
- Suporte a comunicação em grupo
- Mensagens grandes
- Suporta comunicação segura
- Ser usado em internetworks com poucas e confiáveis WANs e LANs

6. COMUNICAÇÃO ENTRE PROCESSOS

Sistemas distribuídos e suas aplicações são compostos de coleções de processos que realizam papéis específicos conforme a natureza de suas tarefas. O papel desempenhado pelos vários processos determina o padrão de comunicação que ocorre entre eles. Um elemento importante de suporte na construção de sistemas distribuídos é disponibilidade de protocolos de alto nível úteis para o suporte dos principais padrões de comunicação que ocorrem em software distribuído assim como facilidades para nomeação e localização de processos.

Este capítulo apresenta tanto construções básicas de comunicação assim como construções mais específicas. Primitivas como *send* e *receive* realizam ações de **passagem de mensagens** entre um par de processos. Sistemas distribuídos podem ser projetados completamente em termos de passagem de mensagens, mas existem certos padrões de comunicação que ocorrem com tal frequência e são tão úteis que devem ser reconhecidos como essenciais de suporte para o projeto e construção de sistemas distribuídos. Dois dos padrões de comunicação mais usados na construção de sistemas distribuídos são: **comunicação cliente/servidor** e **comunicação em grupo**. Outro modelo de comunicação bastante usado na comunicação entre processos é **RPC** (Remote Procedure Call).

6.1 Construções Básicas

Mapeamento de dados em mensagens. Os dados em programas são representados como estruturas de dados enquanto que nas mensagens a informação é sequencial. Desta forma, estruturas de dados devem ser serializadas antes da transmissão e reconstruídas na recepção. Mensagens podem conter dados de diferentes tipos e estes dados (mesmo tipos simples como inteiro, char) podem ser armazenados de forma diferente em diferentes computadores. Assim para que dois computadores quaisquer troquem dados: 1) os dados são convertidos para uma representação externa antes da transmissão e convertidos para a forma local na recepção, 2) na comunicação entre computadores do mesmo tipo, a conversão pode ser omitida e, 3) uma alternativa ao uso da representação externa é transmitir os dados na sua forma nativa anexando um identificador de arquitetura.

Representação externa de dados. XDR (External Data Representation) é um exemplo de padrão que define a representação para dados simples e estruturados (strings, arrays, records), desenvolvido pela SUN para ser usado na troca de mensagens entre clientes e servidores no Sun NFS (Network File System). A figura 3.1 mostra uma mensagem na representação Sun XDR onde a mensagem é uma sequência de objetos de 4-bytes usando uma convenção onde cardinais ou inteiros ocupam 1 objeto. Arrays, estruturas e strings de caracteres são representados como uma sequência de bytes com tamanho especificado. Outra convenção define sobre a parte mais significativa do objeto, qual dos 4 bytes vem primeiro. O uso de um tamanho fixo para cada objeto na mensagem reduz a carga computacional gastando banda passante.

5
“Smit”
“h---”
6
“Lond”
“on--”
1934

Marshalling. O processo de pegar uma coleção de itens de dados e monta-los em uma forma adequada para transmissão em uma mensagem. Unmarshalling é o processo de desmontagem para produzir a coleção de dados equivalente, na chegada. O processo envolve tanto a serialização dos dados quanto a passagem para uma forma de representação externa. Isto pode ser feito a mão (programa explicitamente faz conversão e serialização) ou automaticamente através da geração destas operações com base na especificação dos itens de dados a serem transmitidos. Isto requer uma notação adequada para a descrição dos tipos como por exemplo:

```
Pessoa : TYPE = RECORD [
    Nome, lugar : SEQUENCE OF CHAR;
    Ano : CARDINAL
]
```

6.2 Operações básicas

Passagem de mensagens simples pode ser suportada por duas operações básicas de comunicação por mensagens: *SEND* e *RECEIVE*. Para a comunicação se efetivar um processo envia mensagem para um destino e outro processo no destino recebe a mensagem. Esta atividade envolve a comunicação de dados do processo enviante para o receptor e pode envolver sincronização dos dois processos. O exemplo mostra a definição de um módulo contendo as operações *send* e *receive*.

```
MODULO Mensagem;
EXPORT Mensagem, Erro, Porta, Send, Receive;
```

```
TYPE
    Porta = (* identificador da porta, 128 bits *)
    Mensagem = RECORD
```

```

        Dados : (* seqüência de bytes *);
    END;
    TErro = (NONE, TIMEOUT);

VAR  Erro : TErro;

PROCEDURE Send(p:Porta; m:Mensagem);
    (* Send básico não garante entrega *)

PROCEDURE Receive(p:Porta; VAR m:Mensagem);
    (* Receive básico erro quando timeout *)

END Mensagem.

```

6.3 Comunicação Síncrona e Assíncrona

Uma das propriedades mais importantes das primitivas de troca de mensagens diz respeito a quando a execução das mesmas pode ser retardada (espera). As primitivas são classificadas em síncronas e assíncronas. As primitivas síncronas (bloqueantes) apresentam o seguinte funcionamento: o processo chama a primitiva *send* e especifica um destino e uma mensagem para este endereço. Enquanto a mensagem é enviada, o processo enviante fica bloqueado (sem executar). Da mesma forma, o processo que chama a primitiva *receive* fica bloqueado até que uma mensagem seja recebida, podendo o receptor especificar de quem ele quer receber.

As primitivas assíncronas apresentam o seguinte funcionamento: o processo chama a primitiva *send* e especifica um destino e uma mensagem para ser enviada. Neste caso, o controle retorna ao processo antes que a mensagem seja enviada (não-bloqueante). A vantagem disto é que o processo enviante pode continuar a sua execução em paralelo com a transmissão da mensagem. A operação *receive* na comunicação assíncrona pode ser bloqueante ou não. Entretanto, as primitivas não-bloqueantes apresentam uma desvantagem séria: o espaço de memória ocupado pela mensagem não pode ser modificado até que a mensagem tenha sido enviada. O problema é que o processo enviante não sabe se e quando a transmissão foi feita, sendo assim, ele não sabe quando pode usar o espaço ocupado pela mensagem novamente. Duas soluções são usadas:

- o sistema copia a mensagem para um de seus buffers e deixa o processo continuar (semelhante ao bloqueante, várias cópias podem ser necessárias)
- o enviante é interrompido quando a mensagem foi enviada (sem cópia, dificuldade de tratar interrupções no nível de usuário, race conditions).

A tabela a seguir mostra as variações possíveis na comunicação síncrona e assíncrona.

Tipo de comunicação	Bloqueia no Send	Bloqueia no Receive	Exemplos
Síncrona	Sim	Sim	occam
Assíncrona	Não	Sim	Mach, Chorus, BSD 4.x UNIX
Assíncrona	Não	Não	Charlotte

Da mesma forma que escolhemos entre operações bloqueantes e não-bloqueantes, podemos escolher primitivas com *bufferização* ou *sem bufferização*. No primeiro caso, quando o buffer está cheio, na execução de um *send*, existem duas possibilidades: o *send* fica bloqueado até abrir espaço no buffer ou é retornado uma indicação para o chamador de que a mensagem não pode ser enviada. A situação do *receive* é semelhante. Vamos considerar duas situações extremas: um buffer com capacidade ilimitada e a não existência de buffer. Se o buffer tem capacidade ilimitada o processo que executa um *send* nunca é bloqueado. Se o sistema não tem buffer a execução de um *send* sempre será bloqueada até o correspondente *receive* ser executado (rendezvous). Num sistema baseado em buffer :

- O enviante pode executar várias primitivas *send*.
- É necessário a criação de uma estrutura de buffer por parte do kernel, que deve ser gerenciada.
- A solução implica no envio de uma mensagem para o buffer do receptor (port) onde ela é bufferizada até que requisitada pelo receptor.

6.4 Destino das Mensagens

É necessário definir onde as mensagens vão. Protocolos Internet especificam os destinos com um número de porta usado pelo processo e o endereço Internet do computador no qual ele executa. Com isto o serviço deve sempre executar no mesmo computador. Em sistemas distribuídos prover identificadores independentes de localização é primordial. Quando um destino é especificado como independente de localização, ele deve ser mapeado para um endereço de baixo nível para que a mensagem seja enviada. Sendo assim os serviços podem ser relocados sem informar os cliente da sua localização.

Tipos de destinos. O enviante pode designar um processo fixo como destino (comunicação direta) ou uma localização fixa como recipiente da mensagem (comunicação indireta). Na comunicação direta (endereçamento simétrico), cada processo que deseja enviar ou receber uma mensagem deve explicitamente indicar o nome do destino:

send (P, mensagem) -- envia uma mensagem para o processo P
receive (Q, mensagem) -- recebe uma mensagem do processo Q

Neste caso a comunicação tem as seguintes propriedades:

- um canal de comunicação é estabelecido entre os processos
- um canal de comunicação é associado com exatamente 2 processos
- entre cada par de processos existe apenas um canal de comunicação
- o canal de comunicação é bidirecional

Uma variação deste esquema (endereçamento assimétrico) é quando somente o enviante tem que conhecer o nome do destino:

send (P, mensagem) -- envia uma mensagem para o processo P

receive (anypid, mensagem) -- recebe uma mensagem de qualquer processo

A comunicação direta é fácil de implementar e usar. A desvantagem diz respeito a modularidade, a mudança do nome de um processo faz com que todos os outros processos sejam examinados para uma possível mudança de nomes. Um *receive* em um servidor deveria aceitar mensagens de qualquer cliente. A comunicação direta não permite enviar uma requisição para mais de um servidor.

Na comunicação indireta, um processo cliente não precisa saber o nome do processo servidor mas apenas o local onde serviços podem ser requisitados. As técnicas usadas na comunicação indireta são baseadas em *links* ou *mailboxes*. Os *mailboxes* são baseados em nomes globais, sendo que as mensagens são enviadas/retiradas para/de um *mailbox* :

send (Mailbox, mensagem) -- envia uma mensagem para Mailbox

receive (MailBox, mensagem) -- recebe uma mensagem do MailBox

Neste esquema o canal de comunicação tem as seguintes propriedades:

- pode estar associado a mais de dois processos
- pode existir mais de um canal entre dois processos (*mailbox*)
- pode ser unidirecional ou bidirecional

Associada a cada *mailbox* existe uma fila de tamanho finito onde as mensagens que são enviadas para o mailbox e ainda não foram retiradas permanecem. Qualquer processo que conhece o mailbox pode enviar mensagens. Um mailbox pode pertencer ao processo ou ao SO. No primeiro caso, o mailbox é definido como parte do processo e assim:

- é possível distinguir entre o dono (quem recebe) e o usuário (quem envia)
- quando o processo dono termina o mailbox desaparece
- a propriedade (direitos de acesso) do mailbox pode ser passados para outros processos.

No segundo caso, o mailbox pertence ao SO que fornece mecanismos que permitem o processo:

- criar um mailbox (processo é o dono)
- enviar e receber mensagens
- deletar um mailbox

Quando um processo envia uma mensagem para um mailbox cuja fila esta cheia:

- o processo é suspenso
- o processo é notificado de uma condição de erro
- a mensagem é aceita pelo kernel que providencia seu envio

A tabela a seguir mostra tipos de destinos usados em sistemas operacionais e ambientes de programação distribuída.

Destino da mensagem	Sistema Operacional	Independente de localização?
Processos	V	Sim
Ports	Mach, Chorus e Amoeba	Sim
Sockets	BDS 4.x UNIX	Não
Grupo de processos	V, Amoeba	Sim
Grupo de ports	Chorus	Sim
Objetos	Clouds, Emerald	Sim

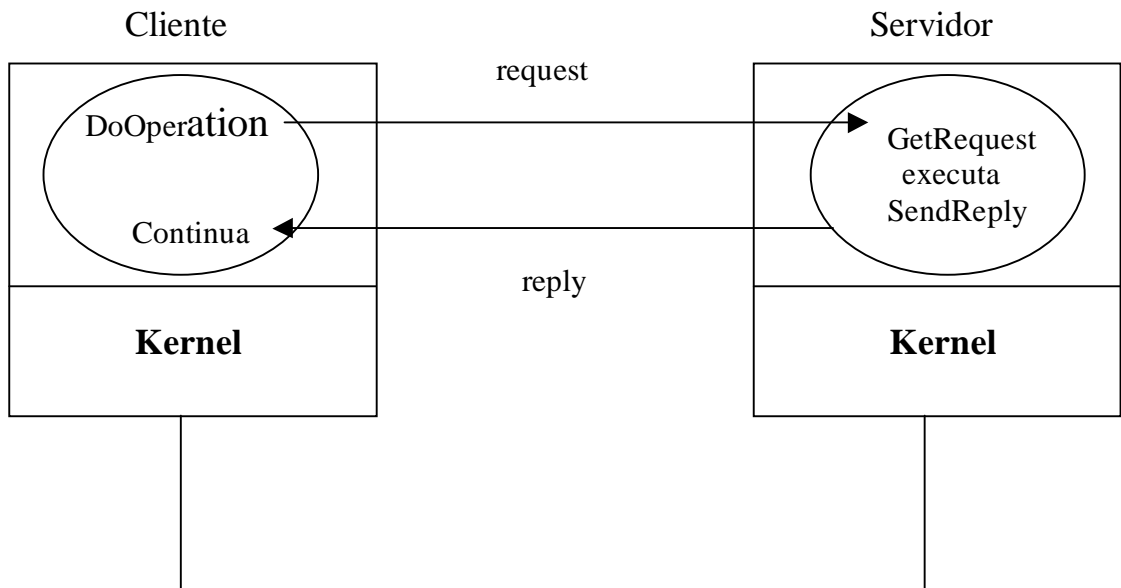
Confiabilidade

O termo mensagem não confiável é usado para referir quando um processo envia uma mensagem sem ack ou reenvio. Por exemplo, usando primitivas bloqueantes, o processo enviante é suspenso até que a mensagem seja enviada. Entretanto, não existe garantia de que, quando ele acorda, a mensagem foi entregue. A mensagem pode ter sido perdida. Existem 3 abordagens possíveis para este problema. A primeira define a semântica do *send* como não-confiável, ou seja, o sistema não dá garantia nenhuma para a entrega das mensagens. Isto fica a cargo dos usuários. A segunda é fazer com que o kernel da máquina que recebe a mensagem envie uma mensagem de reconhecimento do recebimento (Ack) para a máquina enviante. Somente quando este Ack é recebido o processo enviante será liberado. A terceira abordagem utiliza a idéia de que a comunicação cliente-servidor é baseada em mensagens do tipo *request/reply*. Desta forma, a mensagem de *reply* funciona como Ack. Em algumas situações existe um Ack para o *reply*.

Qualquer esquema que envolve o gerenciamento das mensagens requer que cada mensagem deveria ter um identificador único para referencia-la (requisição, enviante).

6.5 Comunicação Cliente-servidor

Com o objetivo de evitar o overhead dos protocolos orientados a conexão (OSI, TCP/IP), o modelo cliente-servidor é baseado num protocolo simples, sem conexão, do tipo *requisição/resposta* (request/reply protocol). O processo cliente envia uma requisição de serviço para um processo servidor. O processo servidor executa o serviço e retorna uma resposta para o processo cliente (dados da requisição ou erro). Duas vantagens podem ser apontadas neste modelo: simplicidade(sem conexão, ack) e eficiência (pilha de protocolos menor). Sistemas como Amoeba, Chorus, Mach e V tem protocolos projetados para suportar este padrão de comunicação e reduzem overhead com um conjunto de 3 primitivas de comunicação: *DoOperation*, *GetRequest* e *SendReply*, mostrado na figura abaixo.



A primitiva *DoOperation* é usada para invocar operações remotas, ela envia uma mensagem para o processo especificado, contendo a requisição e um endereço para receber a resposta. O cliente fique bloqueado até a mensagem ser respondida. A primitiva *GetRequest* é usada pelo servidor para receber requisições de serviço. Depois de executar a requisição o servidor usa a primitiva *SendReply* para enviar a resposta para o cliente.

Confiabilidade do protocolo Cliente-servidor. As mensagens que envolvem requisição e resposta no modelo cliente-servidor podem não chegar ao destino ou o servidor pode falhar e neste caso a recuperação deve ficar a cargo do protocolo cliente-servidor. Para permitir este tratamento a primitiva *DoOperation* deveria usar um timeout quando esta a espera da resposta do servidor. A ação a ser tomada quando um timeout ocorre depende do tipo de protocolo que esta sendo usado. Alguns protocolos cliente-servidor e mensagens enviadas são mostrados na tabela abaixo:

R – Requisição

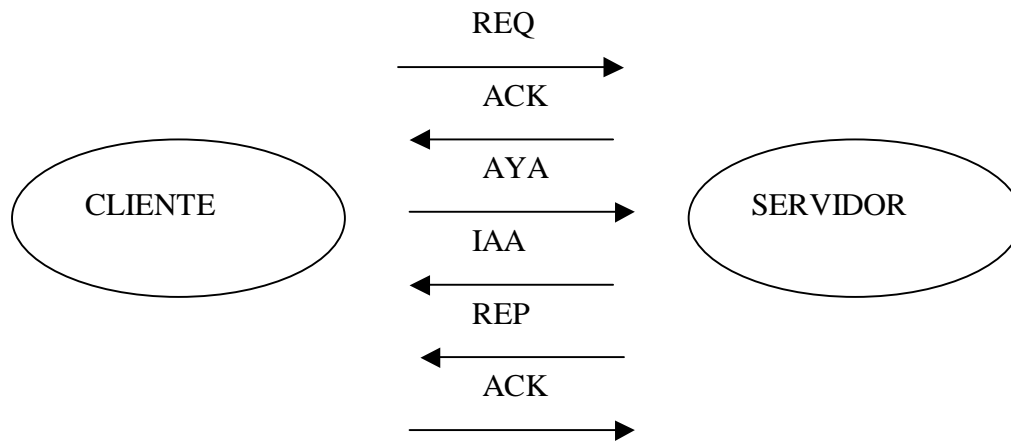
RR – Requisição-resposta

RRA – Requisição-resposta-ack

RARA – Requisição-ack-resposta-ack

Nome	Cliente	Servidor	Cliente	Servidor
R	Requisição			
RR	Requisição	Resposta		
RRA	Requisição	Resposta	Ack-resposta	
RARA	Requisição	Ack-requisição	Resposta	Ack-resposta

Um protocolo mais completo é mostrado na figura abaixo.



6.6 Comunicação em grupo

A comunicação entre pares de processos não é a mais adequada em aplicações onde a comunicação entre um processo e um grupo de processos é necessária. Em sistemas distribuídos isto é usado para prover tolerância a faltas ou melhorar a disponibilidade. Neste caso, mensagens do tipo multicast são mais apropriadas, ou seja, a mensagem é enviada de um processo para os membros de um grupo de processos. Mensagens multicast auxiliam muito na construção de sistemas distribuídos com as seguintes características:

1. Tolerância a falhas baseada em serviços replicados: Um serviço replicado consiste de um grupo de servidores. Requisições são enviadas para todos os membros do grupo. Neste caso, mesmo se algum membro não está disponível o cliente será atendido.
2. Localização de objetos em serviços distribuídos: A localização de objetos (arquivo) em um serviço distribuído (serviço de arquivos) pode ser feita através de mensagens multicast onde só o servidor apropriado responde.
3. Melhor performance através de dados replicados: Para aumentar a performance de um serviço replicas são usadas. Quando os dados são modificados os novos valores são enviados por mensagens multicast para os processos que as gerenciam.
4. Notificação múltipla: Mensagem multicast pode ser usada para avisar um grupo de processos quando algo de interesse acontece.

A partir destas utilizações da comunicação em grupo é possível apontar algumas propriedades úteis para protocolos de comunicação em grupo.

Atomicidade. No caso (1) acima, é necessário que cada servidor receba todas as requisições de forma que todos possam estar no mesmo estado em qualquer instante. Isto requer multicast atômico.

 Multicast atômico: uma mensagem transmitida é recebida por todos os processos membros do grupo ou então não é recebida por nenhum.

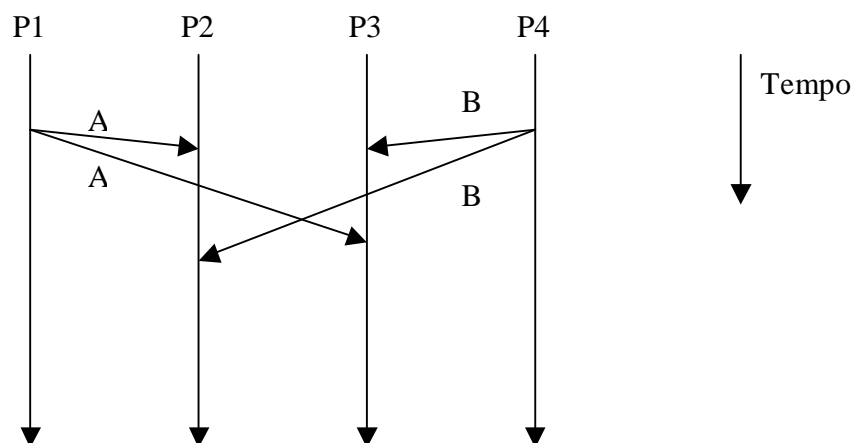
Esta propriedade nem sempre é necessária. Por exemplo, uma vez que a recepção de uma simples resposta é suficiente na comunicação com um grupo de servidores com os mesmos dados, não é necessário assegurar que os processos com as replicas recebam a requisição. Nestes casos multicast confiável é usado.

 Multicast confiável: é um método de transmissão de mensagens com base no melhor esforço de entrega a todos os membros mas, sem garantia. Um multicast não-confiável transmite a mensagem uma vez apenas (Chorus).

No caso (2) é necessário que a mensagem seja recebida pelo que tem a resposta. Multicast confiável é suficiente neste caso, ele pode ser repetido no caso de não haver resposta.

Ordenamento. Multicast atômico e confiável fornecem ordenamento FIFO entre pares de processos. No ordenamento FIFO as mensagens de qualquer cliente a um servidor particular são entregues na ordem de envio. Isto é simples de conseguir anexando números de seqüência as mensagens. Em (1) é necessário que todos os servidores executem suas operações na mesma ordem, considerando que as requisições multicast são transmitidas por diferentes processos.

Sem um mecanismo para assegurar a entrega ordenada das mensagens quando mensagens para o grupo tem origens diferentes ao mesmo tempo, as mesmas podem não chegar na mesma ordem relativa aos membros do grupo.



A forma mais forte de ordenamento é *multicast totalmente ordenado*. Neste caso quando várias mensagens são transmitidas para um grupo as mesmas chegam aos membros do grupo na mesma ordem. Ordenamento total aplica-se a todas as mensagens enviadas a um grupo e também tem um custo alto de comunicação.

Gerência de Grupos. Na comunicação em grupo é necessário de alguma forma gerenciar a criação e remoção de grupos assim como permitir que processos juntem-se aos grupos e retirem-se dos mesmos. Uma abordagem é através de um servidor de grupos para o qual todas estas requisições são enviadas. O servidor mantém uma base de dados completa de todos os grupos e seus componentes.

Outra abordagem é gerenciar os grupos de forma distribuída. Para entrar no grupo um processo envia uma mensagem para o grupo anunciando sua presença, para sair o processo envia uma mensagem de adeus para todos do grupo. Até aqui parece simples mas duas características tornam as coisas mais difíceis: 1) quando um membro quebra ele efetivamente deixa o grupo – neste caso não existe anuncio do fato e os outros membros tem que descobrir isto; 2) juntar-se e deixar o grupo deve ser sincronizado com as mensagens sendo enviadas.

Implementação de comunicação em grupo. A forma mais simples de multicast é o multicast não-confiável no qual uma mensagem é enviada para cada destino. Uma implementação simples pode ser baseada no envio da mensagem para a porta de cada componente do grupo.

```
PROCEDURE multicast (destino: ARRAY OF Portas; m: Mensagem);  
VAR i:INTEGER;  
BEGIN  
  FOR i:= 0 TO N(destinos) DO Send(destino(i), m) END;  
END;
```

A *eficiência* da comunicação em grupo pode ser melhorada na maioria das redes locais (onde é possível mensagens do tipo broadcast). Algumas redes também permitem o envio para um grupo de computadores da mesma (multicast). No caso dos processos pertencentes a um grupo estarem localizados em uma rede local uma mensagem broadcast é suficiente para atingir todos os membros do grupo. Entretanto, existe a possibilidade do sistema ser composto por várias redes. Neste caso o mecanismo deveria prover serviço atingindo múltiplas redes locais e endereçar processos independente da localização.

A implementação acima é potencialmente não-confiável no que diz respeito possibilidade de a mensagem não chegar a todos os destinos. Existem duas razões para uma mensagem chegar a alguns destinos e a outros não: uma das mensagens pode ser descartada, o processo transmissor pode falhar depois de mandar algumas mas não todas mensagens. Isto não é aceitável para aplicações que requerem multicast atômico ou totalmente ordenado.

Uma forma de implementar multicast confiável é o enviante da mensagem esperar a recepção de ack de cada destino, quando receber todos ele completa o multicast podendo informar ao chamador que mandou para todos os membros do grupo. No caso de não receber alguns acks ele pode retransmitir até que ele assume que um determinado membro falhou e é removido do grupo. Entretanto, se o enviante falha durante este processo o multicast não será atômico a não ser que o protocolo assegure que um dos recebedores assumirá o papel do enviante. Cada membro do grupo que recebe uma mensagem e retorna um ack depois monitora a origem para detectar falha ou confirmação do multicast. Os

membros do grupo podem saber quando o multicast terminou através de uma notificação da origem (novo multicast). Este protocolo tem desempenho fraco, mesmo em situação normal.

7. CHAMADA REMOTA DE PROCEDIMENTO (RPC)

O relacionamento cliente/servidor pode ser representado pelo mecanismo de chamada de procedimento (o chamador transfere o controle para o chamado e assim que o chamado termina o controle retorna para o chamador). Considerando que o procedimento chamado pode estar em uma máquina diferente, quando um processo da máquina A chama um procedimento na máquina B, o processo chamador é suspenso e a execução do procedimento é realizada na máquina B. As informações são transportadas do chamador para o chamado através de parâmetros e voltam no resultado da chamada. Este método é conhecido por chamada remota de procedimento (RPC).

A meta de um mecanismo de chamada remota de procedimento é o de manter o máximo possível a semântica da chamada de procedimento convencional em um ambiente de implementação completamente diferente. A definição de um procedimento remoto especifica parâmetros de entrada e saída da mesma forma que na chamada convencional. Em RPC os parâmetros de entrada são enviados para o servidor através da passagem de mensagem e os parâmetros de saída são enviados ao cliente na mensagem de resposta. Parâmetros de entrada são equivalentes a passagem por valor. A passagem de parâmetros por referência necessita informações como se o parâmetro é de entrada, saída ou ambos. A especificação destas alternativas normalmente é feita através do uso de uma linguagem de definição de interface (IDL), essencial para um sistema de RPC.

7.1 Operação básica do mecanismo RPC

Vamos primeiro verificar como uma chamada de procedimento convencional funciona para que possamos entender como funciona o mecanismo RPC.

Considere a chamada *count = read(fd, buf, nbytes)*. Para fazer a chamada o chamador coloca os parâmetros na pilha em ordem, ultimo primeiro. Depois de terminado o procedimento coloca o valor de retorno em um registrador, remove o endereço de retorno e transfere o controle de volta para o chamador.

No caso da utilização do mecanismo RPC a transparência da chamada é conseguida de maneira análoga. Quando *read* for um procedimento remoto, uma versão diferente deste procedimento, *stub cliente*, é colocado na biblioteca e é chamado da mesma forma que o convencional e causa um *trap* para o SO. A chamada de um procedimento remoto segue os seguintes passos :

1. O cliente chama o *stub cliente* como uma chamada normal
2. O *stub cliente* é responsável pela construção de uma mensagem e chama o kernel
3. O kernel do chamador envia a mensagem para o kernel do chamado
4. O kernel do chamado recebe a mensagem e passa a mesma para o *stub servidor*
5. O *stub servidor* retira parâmetros da mensagem e faz uma chamada para o servidor
6. O servidor completa o trabalho e retorna para o *stub servidor* passando os resultados
7. O *stub servidor* coloca os resultados em uma mensagem e chama o kernel
8. O kernel do chamado envia a mensagem para o kernel chamador

9. O kernel do chamador recebe a mensagem e passa a mesma para o *stub cliente*
10. O *stub cliente* retira os resultados e retorna para o cliente

Os parâmetros podem ser passados por valor ou por referência. No caso dos parâmetros por valor o *stub* copia os parâmetros para a mensagem e envia. Entretanto a passagem de parâmetros por referência (pointers) é feita, quando é, com muita dificuldade. Se em uma chamada *read* existir um parâmetro que é o endereço do buffer (1000) ele só é significativo para o cliente e não para o servidor. Soluções :

- proibir a utilização de parâmetros por referência
- copiar o buffer (sabe-se o tamanho) na mensagem (copy/restore)
- buffer é de input para o servidor, não manda de volta; output cliente não precisa enviar

7.2 Binding

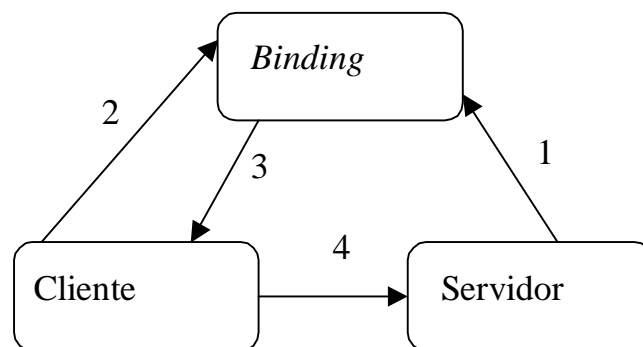
Como o cliente localiza o servidor? Uma forma é manter o endereço físico do servidor no cliente. Entretanto, isto nos traz uma série de problemas (ligação estática). Para evitar estes problemas alguns SOD's utilizam ligação dinâmica (dynamic binding) para fazer o casamento entre cliente e servidor. Como ponto fundamental para a ligação dinâmica esta a especificação formal do servidor que especifica o nome do servidor (servidor_de_arquivos), a versão (3.1) e uma lista de serviços fornecidos pelo servidor (read, write, create, delete). Cada uma destes serviços é especificado em termos dos parâmetros e tipos :

```
#include <header.h>
especificação do servidor_de_arquivos versão 3.1
long read ( in char nome[MAX_P], out char buf[BUF_T], in long bytes, in long posição );
long write ( in char nome[MAX_P], in char buf[BUF_T], in long bytes, in long posição );
int create ( in char nome[MAX_P], in int modo );
int delete ( in char nome[MAX_P] );
fim;
```

Esta especificação é inicialmente utilizada no gerador de stubs, tanto do cliente quanto do servidor, que são colocados na biblioteca. Desta forma, quando um programa de usuário (cliente) requisita qualquer serviço definido na especificação, o stub cliente é ligado com o programa. Da mesma forma isto acontece com o servidor.

A ligação dinâmica funciona da seguinte forma (figura utilização do binder):

1. O servidor exporta a sua interface (registra-se no binder)
2. O cliente, na sua primeira chamada, requisita um servidor_de_arquivos adequado
3. O binder (servidor de nomes) responde com o identificador (endereço) ou negativa
4. Cliente requisita serviço para o servidor



Este método é bastante flexível :

- pode tratar múltiplos servidores utilizando a mesma interface
- o binder pode policiar os servidores e atuar como autenticador

A tabela abaixo mostra uma possível interface para o binder.

Chamada	Entrada	Saída
Register	Nome, versão, tratador, identificador	
Deregister	Nome, versão, identificador	
Look up	Nome, versão	Tratador, identificador

A desvantagem, entretanto, diz respeito a possibilidade de o binder se tornar o ponto crítico do sistema (gargalo) devido o fato de que vários processos clientes podem estar tentando acessar o binder. Neste caso, é possível utilizar múltiplos binders.

7.3 A semântica das RPC's

A meta das RPC's é esconder as comunicações fazendo com que elas pareçam como as chamadas locais. Entretanto, o fato de o chamador e o chamado serem processos distintos que executam em máquinas diferentes existe a possibilidade de falhas dos processos, dos computadores ou do sistema de comunicação. O procedimento remoto pode não ser completado com sucesso, ou seja, a mensagem com o resultado pode não retornar para o chamador devido a um conjunto de eventos. Estes eventos formam a base para o projeto da semântica das RPC's :

1. O cliente não consegue localizar o servidor
2. A requisição de serviço é perdida
3. A resposta do servidor é perdida
4. O servidor cai (depois de receber uma requisição)
5. O cliente cai (depois de enviar uma requisição)

O cliente não consegue localizar o servidor. O cliente pode estar desatualizado e não conseguir localizar o servidor adequado (a versão do servidor é mais atual daquela do cliente). Neste caso, teremos uma falha que deve ser tratada:

- retornar um código de erro na chamada – retornar -1 para indicar a existência de erro, em UNIX uma variável global *errno* contém o tipo de erro. Esta solução não é geral, em alguns casos o valor -1 pode ser um valor normal (escolha do código).
- tratamento de excessão – algumas linguagens fornecem a possibilidade de o usuário escrever procedimentos especiais que são invocados (nem todas linguagens). Escrever procedimentos especiais neste caso tira a transparência.

A requisição de serviço é perdida. Neste caso, o tratamento requer um timer que é iniciado pelo kernel quando a requisição é enviada. Se o tempo termina antes que seja recebida uma resposta ou Ack, o kernel envia novamente a requisição. Se a mensagem foi realmente perdida o servidor não tem como diferenciar uma retransmissão e a primeira. Quando várias mensagens são perdidas o kernel pode desistir e concluir, erradamente, que o servidor esta fora, neste caso voltamos a primeira situação.

A resposta do servidor é perdida. Da mesma forma que o anterior, o tratamento pode ser o envio da requisição novamente. O problema é que o kernel (cliente) não sabe porque não obteve resposta, a requisição ou a resposta pode ter se perdido ou o servidor está lento. Algumas operações podem ser repetidas sem problemas, ou seja, podem ser executadas quantas vezes necessário sem causar problemas (idempotentes). Por exemplo, uma operação para adicionar um elemento em um conjunto é idempotente porque vai sempre causar o mesmo efeito. Entretanto, uma operação para incluir um item em uma sequência não é idempotente porque a sequência vai ser aumentada cada vez que a operação for realizada. Neste caso, estruturamos todas as requisições de uma forma idempotente (o que não é conseguido na prática) ou atribuímos a cada requisição um número de sequência que vai permitir ao servidor diferenciar entre uma requisição original e uma retransmissão. Para melhorar, podemos adicionar um bit na mensagem para distinguir requisições originais de retransmissões.

O servidor cai. Este problema está relacionado com idempotência mas não pode ser resolvido com número de sequência. O problema é quando o servidor cai. Ele pode receber a requisição executá-la e antes de responder cair. Ele pode receber a requisição e logo em seguida cair. No primeiro caso o sistema tem que notificar a falha para o cliente, enquanto que no segundo caso basta o cliente retransmitir. O problema é que o kernel do cliente não sabe dizer qual deles aconteceu. Neste caso três abordagens podem ser adotadas:

- pelo menos uma vez : tenta até receber pelo menos resposta, quando uma RPC é terminada o cliente não saberá quantas vezes chamou o servidor. Se o servidor pode ser projetado com operações idempotentes esta solução é aceitável.
- no máximo uma vez : desiste imediatamente e relata o erro, se a primeira tentativa não obteve sucesso, desiste e relata o insucesso (mais usada, operações não idempotentes).
- talvez : não garante nada , o cliente não sabe dizer com certeza se o procedimento foi chamado ou não. Geralmente não aceito.

Nenhuma delas é muito atrativa, a mais apropriada seria exatamente uma vez, ou seja, chama uma vez e é executado. Entretanto, este tipo de solução não tem como ser garantida. Em outras palavras, a possibilidade do servidor quebrar muda radicalmente a natureza do RPC e faz distinções claras entre sistemas distribuídos e de um único processador.

O cliente cai. O que acontece se um cliente envia uma requisição para um servidor e cai antes do servidor responder? Uma computação está ativa e nenhum pai está esperando pelo resultado, este tipo de computação é chamada orfã. Este tipo de computação pode causar vários problemas desde gastar CPU, bloquear arquivos e recursos até a situação onde o cliente volta e executa a mesma RPC e logo em seguida a resposta do antigo chega logo em seguida. O que fazer a respeito das computações orfãs? Algumas soluções são propostas:

- Exterminar – antes de mandar a mensagem RPC o cliente (stub) registra a ação em um arquivo de log que é mantido em disco. Quando volta o log é verificado e a computação orfã é explicitamente removida. Algumas desvantagens desta abordagem são : operação log em cada RPC, orfãs podem fazer RPC.
- Reincarnar – divide o tempo em épocas, quando o cliente volta ele envia um broadcast para todas máquinas anunciando uma nova época. Esta mensagem faz com que todas computações remotas sejam removidas. Mesmo que algumas sobrevivam quando responderem a época será diferente.
- Reincarnar gentilmente – uma variação é tentar encontrar o dono da computação e só remove-la se o mesmo não for encontrado.
- Expiração – cada RPC tem um tempo T para executar, se não conseguir ele deve requisitar outro tempo. Depois da queda, se o cliente esperar um tempo T antes de iniciar todos os orfãos não existirão mais. Problema, quanto vale T?

Na pratica nenhum destes métodos é desejável visto que a remoção das computações orfãs traz conseqüências nem sempre previstas.

7.4 Implementação

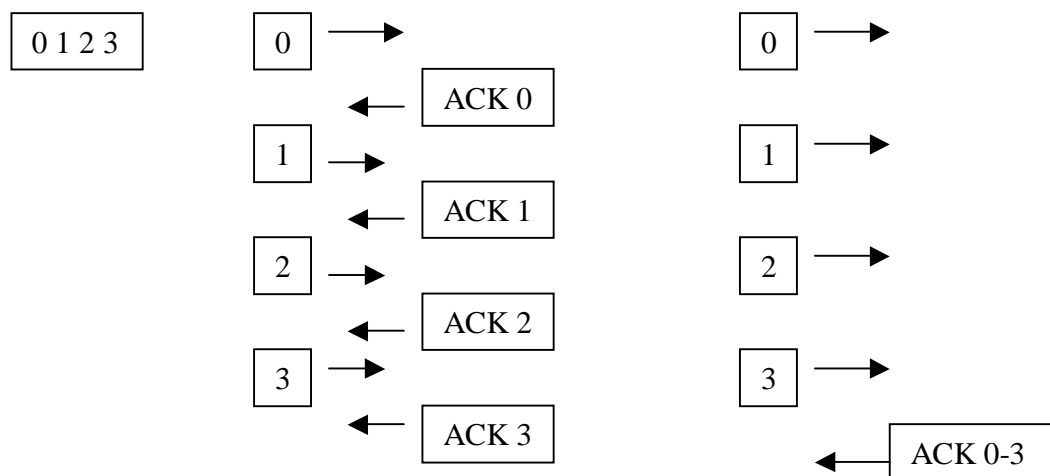
O sucesso ou falha de um sistema distribuído depende da performance. A performance por sua vez, é dependente da velocidade da comunicação. Algumas considerações de implementação para RPC enfatizando a performance e onde o tempo é gasto são feitas a seguir.

Protocolos RPC - De uma maneira geral qualquer protocolo, desde que transfira bits do cliente para o servidor, pode ser usado. Entretanto, existem algumas escolhas a fazer:

- a) protocolo orientado a conexão, sem conexão – O uso de um protocolo orientado a conexão é benéfico porque a partir do estabelecimento da conexão toda a comunicação usará esta conexão, a comunicação é considerada segura. Em redes de longa distância isto é muito importante. Entretanto, isto causa (comparado com LAN) causa uma perda de performance. Por isto, a maioria dos sistemas distribuídos implementados em curtas distâncias usam protocolos sem conexão.

- b) protocolo de propósito geral ou específico – Como não existe um padrão para protocolos RPC usar um específico significa projetar o seu próprio. Alguns sistemas utilizam IP ou UDP como protocolo básico e a maior vantagem diz respeito a aceitação e confiabilidade do mesmo. O problema é a performance, visto que IP não foi projetado para ser um protocolo para RPC.
- c) tamanho da mensagem e pacote – Um RPC tem um grande e fixo overhead além dos dados a transmitir. Assim ler um arquivo de 64K em um único RPC é mais eficiente do que 64 RPCs. Para isto é importante que o protocolo e a rede permitam transmissões grandes. Alguns sistemas RPC são limitados a mensagens pequenas (Sun-8K) e muitas redes não tratam pacotes grandes (Ethernet- 1536bytes) assim um RPC tem de se quebrado em vários pacotes causando overhead extra.

Reconhecimento. Quando existe a necessidade de dividir um RPC em vários pacotes, como será o reconhecimento destes um por um ou não? Considere o exemplo em que um cliente deseja escrever um bloco de dados de 4K em um servidor de arquivos mas o sistema trata com pacotes de até 1k no máximo. Uma estratégia seria o cliente enviar um pacote e esperar por ack depois enviar outro e novo ack e assim por diante, protocolo *stop-and-wait*. Outra alternativa é ele simplesmente enviar todos os pacotes o mais rápido possível e depois receber ack da mensagem toda, protocolo *blast* (rajadas). A figura a seguir mostra o funcionamento das duas alternativas.



Cópias. A necessidade de fazer cópias em um RPC envolve um tempo que pode ser considerado alto. Em sistemas onde o espaço de endereçamento do kernel e do usuário são disjuntos a cópia é necessária. O número de vezes que uma mensagem deve ser copiada varia de 1 a 8 vezes, dependendo do hardware, software e do tipo de chamada.

No melhor caso, DMA (Direct Memory Access) é usado para o kernel copiar mensagem do espaço de endereçamento do stub cliente para a placa de rede (copia 1), depositando a mesma no kernel servidor (a interrupção de chegada do pacote ocorre em poucos microssegundos). Após o kernel inspeciona o pacote e mapeia a página que o contém para o

espaço de endereço do servidor. Se este tipo de mapeamento não é possível, o kernel copia o pacote para o stub servidor (copia 2).

No pior caso, a seguinte sequência pode ser necessária :

1. kernel copia a mensagem do stub cliente para buffer do kernel para transmissão (não conveniente copiar direto ou rede esta ocupada);
2. depois o kernel copia a mensagem para um buffer de hardware na placa de rede;
3. neste ponto ocorre a transferência da mensagem para um buffer de hardware no destino.
4. quando a interrupção de chegada de pacote ocorre na máquina servidora o kernel copia a mesma para um buffer do kernel (não sabe ainda onde coloca-la);
5. agora a mensagem tem de ser copiada para o stub servidor.

Mais 3 cópias podem ser necessárias se a chamada tem um array passado por valor. O array tem de ser copiado para a pilha do cliente para chamar o stub, da pilha para o buffer da mensagem durante o marshaling e da mensagem recebida no stub servidor para a pilha do servidor antes da chamada do mesmo.

Soluções para evitar as cópias normalmente envolvem características de hardware. Uma delas é conhecida como *scatter-gatter*. Uma placa de rede que tem esta característica pode ser inicializada para montar um pacote através da concatenação de 2 ou mais buffers de memória. Isto permite eliminar cópias. Da forma similar, esta característica permitiria separar partes do pacote (na chegada) em diferentes buffers. De uma forma geral, eliminar cópias no lado enviante é mais fácil do que no lado recebedor.

Temporizadores. Todos protocolos consistem na troca de mensagens em um meio de comunicação e na maioria dos sistemas mensagens podem ser perdidas devido a ruídos ou sobreposição. Desta forma, na maioria dos casos os protocolos usam um temporizador sempre que uma mensagem é enviada e uma resposta esperada. Se a resposta não é recebida dentro de um tempo esperado, o temporizador faz com que a mensagem seja re-transmitida. O tempo de máquina que é gasto no gerenciamento de temporizadores não deveria ser subestimado. Um temporizador requer uma estrutura de dados que especifica quando ele expira e o que fazer quando isto acontece. Esta estrutura de dados é inserida em uma lista contendo outros temporizadores pendentes (mantida em ordem de tempo). Quando um ack ou resposta chega antes do temporizador expirar, a entrada correspondente deve ser retirada da lista.

Uma forma mais eficiente de tratar temporizadores pode ser associar os mesmos as informações mantidas sobre os processos pelo kernel. Desta forma, ao invés de armazenar estes valores em listas cada entrada da tabela de processos tem um campo para temporizadores. Para inicializar um temporizador soma o tempo atual com o timeout e armazena na tabela de processos. Desligar o temporizador implica em zerar este campo. Para este método funcionar, periodicamente o kernel verifica os valores na tabela com o valor atual. Qualquer valor igual ou menor que o tempo atual corresponde a um temporizador expirado.

8. ESTUDOS DE CASOS

8.1 UNIX Sockets

As primitivas de comunicação entre processos em sistemas UNIX (BSD 4.x) são chamadas de sistema implementadas como um nível suportado por protocolos TCP e UDP. Os destinos das mensagens são especificados como endereços de sockets, um identificador de comunicação que consiste de um número de porta local e um endereço internet. Estes endereços podem ser passados em duas direções: do processo para o kernel e do kernel para o processo. Funções de conversão de endereço convertem representação textual de um endereço para valores binários. A maioria das funções sockets utilizam um endereço de socket que é representado por uma estrutura. Por exemplo, utilizando Ipv4 a estrutura é conhecida pelo nome *sockaddr_in* (netinet/in.h) assim definida:

As operações de comunicação entre processos são baseadas em pares de sockets e consiste de troca de informações através da transmissão de mensagem entre um socket de um processo e o socket de outro processo. As mensagens são enfileiradas no socket do enviante até a transmissão e recepção de ack (se requerido). Na chegada as mensagens são enfileiradas no socket receptor até que o processo receptor executa a chamada adequada para recepção. Qualquer processo pode criar um socket para comunicar-se com outro processo. A chamada da função *socket* cria um socket com características que definem o domínio da comunicação (UNIX, Internet) e o tipo (datagrama, stream).

Stream. Para usar este tipo de comunicação os processos primeiro tem de estabelecer uma conexão. Uma vez estabelecida a conexão eles podem trocar mensagens nas duas direções. Dados disponíveis são lidos na ordem em que foram escritos. Se não existem dados para ler o receptor fica bloqueado, o enviante fica bloqueado quando o buffer esta cheio. Normalmente, no modelo cliente/servidor o servidor escuta e aceita uma conexão e depois cria um novo processo para tratar as requisições do cliente.

A comunicação envolve os seguintes passos:

- servidor cria um socket
- servidor estabelece endereço para o socket
- servidor escuta pedidos de conexão
- servidor aceita conexão
- cliente cria socket
- cliente pede conexão
- servidor e cliente enviam e recebem mensagens

Este tipo de comunicação é usado em aplicações do tipo rlogin, rsh, onde o nome do computador remoto é fornecido como argumento. Qualquer computador que pode aceitar estas requisições atua como servidor, escutando o cliente e aceitando a conexão.

Datagrama. Neste tipo de comunicação o socket é identificado cada vez que a comunicação é feita. Para isto o processo enviante usa o seu descritor local de socket e o endereço do socket receptor cada vez que ele envia uma mensagem.

Este tipo de comunicação envolve os seguintes passos:

- os dois processos criam sockets e recebem um descritor
- processo enviante estabelece endereço do socket (qualquer disponível)
- processo receptor estabelece endereço do socket (conhecido pelo enviante)
- processo enviante envia mensagem (endereço conhecido)
- processo receptor recebe mensagem

Primitivas. A comunicação via sockets é feita através de funções disponíveis para o usuário. Estas funções são necessárias no desenvolvimento de aplicações cliente/servidor. Como funções elementares podemos citar:

- *socket(int family, int type, int protocol)* – a primeira coisa que um processo deve fazer para uma comunicação é chamar esta função, especificando o tipo de comunicação. *Family* especifica família de protocolos (AF_INET, AF_UNIX) e *type* especifica o tipo de comunicação (SOCK_STREAM, SOCK_DGRAM).
- *Connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen)* – esta função é usada para um cliente TCP estabelecer conexão com um servidor TCP. *Sockfd* é o descritor retornado na chamada anterior, os outros dois argumentos especificam endereço do socket e o tamanho da estrutura. A estrutura do endereço deve conter o endereço IP e o número da porta do servidor. A função retorna quando a conexão é estabelecida ou ocorre um erro.
- *Bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen)* – esta função atribui um endereço local para o socket. No caso de protocolos Internet o endereço é uma combinação do endereço IP (v4-32bits, v6-128bits) e um número de porta TCP ou UDP (16-bits). Um processo pode associar um endereço IP específico para seus sockets. O endereço IP deve pertencer a uma interface do host. Usando TCP a chamada desta função permite especificar um número de porta, um endereço IP, ambos ou nenhum.
- *Listen (int sockfd, int backlog)* – esta função só é chamada para um servidor TCP e faz com que conexões (uma ou mais) possam ser aceitas neste socket. A função é chamada antes da função que aceita a conexão. O segundo argumento da função a princípio indica quantas conexões podem ser aceitas.
- *Accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen)* – função chamada para o servidor TCP retornar a próxima conexão completada. Se a fila de conexões é vazia o processo é bloqueado. Os argumentos *cliaddr* e *addrlen* são usados para retornar o endereço de protocolo do processo conectado (cliente). Quando bem sucedida esta função retorna um novo descritor criado pelo kernel, que descreve a conexão TCP com o cliente.

- *Close (int sockfd)* – esta função fecha um socket e termina a conexão TCP.
- *Read (int sockfd, void *buff, size_t nbytes)* – esta função recebe mensagens do socket.
- *Write(int sockfd, const void *buff, size_t nbytes)* esta função envia mensagens para o socket.
- *Recvfrom (int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr *from, socklen_t * addrlen); sendto(int sockfd, const void *buff, size_t nbytes, int flags, struct sockaddr *to, socklen_t * addrlen)* – estas funções são similares a read e write com a necessidade de argumentos adicionais. Os 3 primeiros são idênticos. O argumento *to* na função *sendto* é uma estrutura de endereço socket que contém o endereço (IP, porta) do destino da mensagem. Na função *recvfrom* a estrutura apontada por *from* é completada com o endereço do fonte da mensagem. Os 2 últimos argumentos de *recvfrom* são similares aos de *accept*. Da mesma forma, *sendto* tem os argumentos similares a *connect*.

8.2 ISIS

Como exemplo de um sistema de comunicação em grupo vamos dar uma olhada no sistema ISIS desenvolvido na Universidade Cornell. ISIS é uma ferramenta para construção de aplicações distribuídas, não é um sistema operacional mas sim um conjunto de programas que executam no sistema UNIX ou outros sistemas operacionais. O estudo deste sistema é possível pela boa documentação existente.

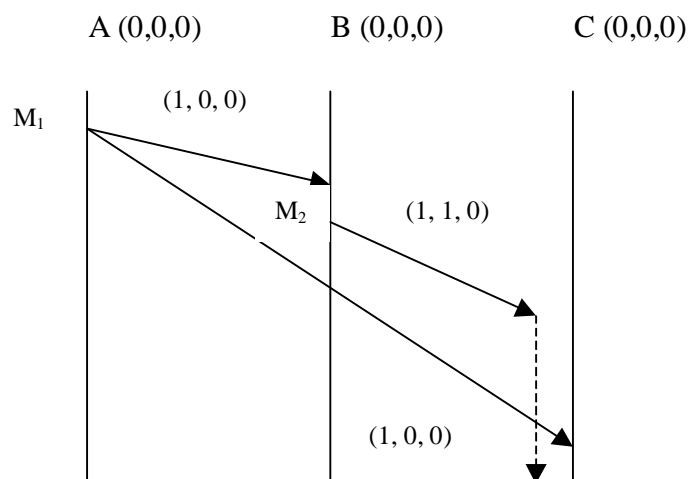
A idéia chave do sistema ISIS é o sincronismo sendo que as primitivas de comunicação permitem formas diferentes de broadcast atômico. Diferentes formas de sincronismo são tratadas. Um sistema completamente síncrono é aquele no qual os eventos acontecem de forma estritamente seqüencial. Por exemplo, se um processo A envia uma mensagem para os processos B, C e D, as mensagens chegam instantaneamente em todos os destinos. Entretanto, este tipo de sincronismo é impossível de ser construído. Assim outros tipos, com requisitos mais fracos, são estabelecidos. Sincronismo fraco, é aquele onde os eventos tem um tempo associado mas aparecem na mesma ordem para todos os participantes. Os processos recebem as mensagens na mesma ordem. Sincronismo virtual, é aquele onde a restrição de ordenamento é relaxada. Isto significa que se duas mensagens tem relacionamento causal – processo A envia uma mensagem para B que verifica a mesma e envia uma mensagem a C, a segunda mensagem tem relação causal com a primeira – todos os processos devem recebe-las na mesma ordem. Se elas são concorrentes – A envia uma mensagem para B e no ao mesmo tempo C envia uma para D – nenhuma garantia é feita, o sistema esta livre para entregar em ordem diferente.

Primitivas de comunicação. As primitivas que permitem broadcast em ISIS são: ABICAST, CBCAST e GBCAST. ABICAST fornece comunicação com sincronismo fraco e é usada para transmitir dados para os membros do grupo. CBCAST fornece comunicação

com sincronismo virtual e também é usada para enviar dados. GBCAST é similar a ABCAST porém é usada para gerenciar os grupos e não para enviar dados.

ABCAST – o enviante, A, atribui um timestamp (numero de sequência) para a mensagem e envia para todos os membros do grupo. Cada um pega o seu próprio timestamp, maior que qualquer outro que tenha sido enviado ou recebido, e envia de volta para A. Quando todos chegam, A escolhe o maior e envia uma mensagem *commit* para todos os membros do grupo contendo o timestamp. Mensagens *commit* foram entregues para os programas de aplicação na ordem dos timestamps. Pode ser mostrado que este protocolo garante que todas mensagens serão entregues a todos os processos na mesma ordem. O protocolo também é complexo e caro.

CBCAST – garante entrega ordenada apenas para mensagens que tem relação causal. Se um grupo tem n membros, cada processo mantém um vetor com n componentes, um por membro do grupo. O i ésimo componente deste vetor é o numero da última mensagem recebida em sequência do processo i . Os vetores são gerenciados pelo sistema de runtime não pelos processos, e são inicializados em 0. Quando um processo tem uma mensagem para enviar, ele incrementa o seu próprio slot no seu vetor como parte da mensagem. A figura abaixo mostra um exemplo.



Quando M1 chega em B uma verificação de dependência é feita. O primeiro componente do vetor é maior que o de B e os outros são iguais, a mensagem é aceita e passada para o membro do grupo executando em B. Se qualquer outro componente do vetor fosse maior do que o de B a mensagem não seria entregue. Agora B envia uma mensagem para C, a mesma chega antes de M1. O vetor indica que B já recebeu uma mensagem de A antes de enviar M2 e uma vez que C ainda não recebeu nada de A, M2 será retardada até que a mensagem de A chegue.

O algoritmo para decidir quando passar uma mensagem para o usuário ou retardar-la tem o seguinte funcionamento:

$V_i = \text{Mens.VETOR}[i]$

$L_i = \text{Local.VETOR}[i]$

Mensagem enviada por j

1ª condição para aceitação: $V_j = L_j + 1$

2ª condição para aceitação: $V_i \leq L_i$ para todos $i <> j$.

8.3 Sun RPC

Sun RPC é fornecido como parte do sistema operacional Sun UNIX e também é disponível para outras instalações NFS. Os implementadores podem escolher usar chamadas remotas com UDP/IP ou TCP/IP. O sistema fornece uma linguagem de interface (IDL) que é uma extensão de XDR e um compilador de interface chamado *rpcgen*. A linguagem é usada para especificar uma interface que contém um número de programa e um número de versão com definição de procedimentos e definição de tipos de dados. A seguir um exemplo de especificação de uma interface RPC.

```
1    struct square_in {
2        long    arg1;
3    };
4    struct square_out {
5        long    res1;
6    };
7    program SQUARE_PROG {
8        version SQUARE_VERS {
9            square_out SQUAREPROC(square_in) = 1;
10        } = 1;
11    } = 0x31230000;
```

1 – 6 definição de duas estruturas uma para os argumentos de entrada e outra para resultado. 7 – 11 definição de um programa RPC de nome SQUARE_PROG que consiste de uma versão contendo um procedimento. O argumento deste procedimento é do tipo square_in e o valor de retorno é do tipo square_out. São atribuídos valores para o procedimento (1), para a versão (1) e para o programa (32-bits hexadecimal). Esta especificação é compilada usando o compilador *rpcgen*. A seguir mostramos o programa cliente que vai fazer uso do procedimento remoto.

```
1    #include    "unpipc.h"
2    #include    "square.h"
3    int
4    main(int argc, char **argv)
5    {
6        CLIENT *c1;
7        square_in in;
8        square_out outp;
9        if (argc != 3)
10            err_quit("uso: client <nome_do_host> <valor_inteiro>");
11        c1 = Clnt_create (argv[1], SQUARE_PROG, SQUARE_VERS, "tcp");
12        in.arg1 = atol(argv[2]);
13        if ( (outp = squareproc_1(&in, c1)) == NULL)
14            err_quit("%s", clnt_sperror(c1, argv[1]));
15        printf("result: %ld\n", outp->res1);
16        exit(0);
17    }
```

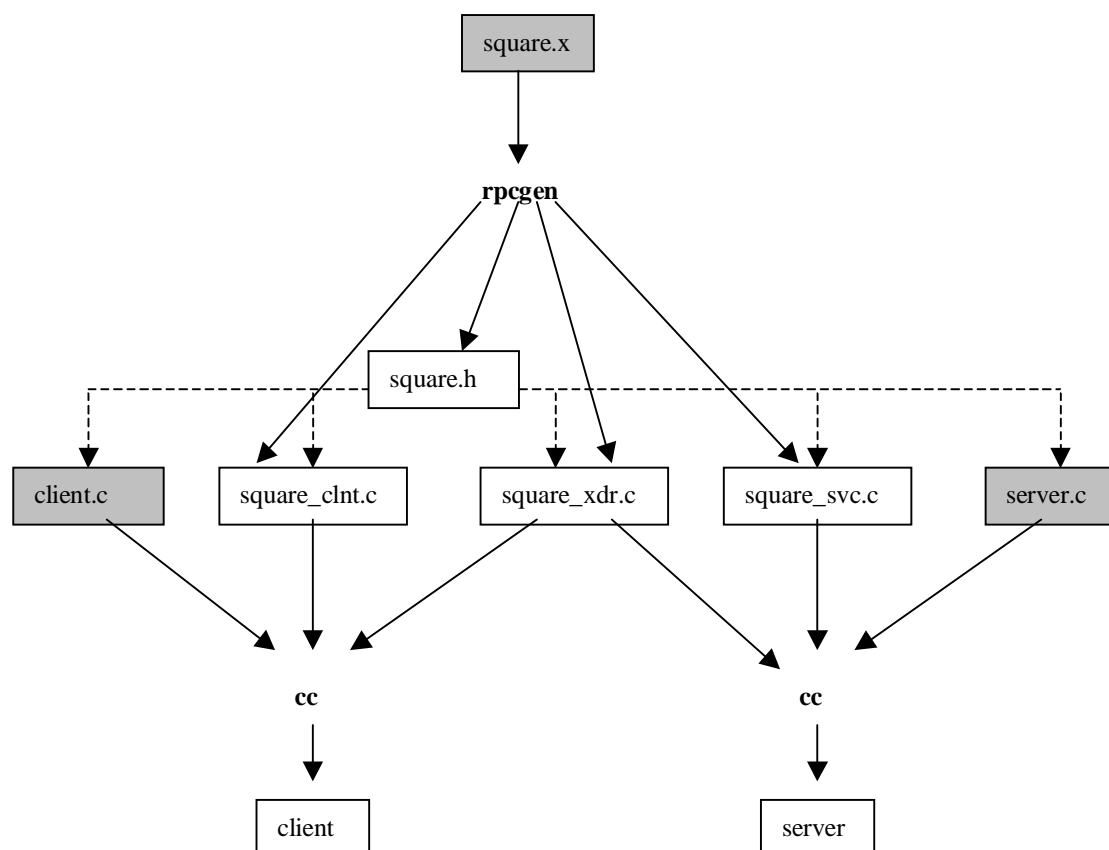
Linha 6 declara o tratador do cliente que retornado na chamada na linha 11. Este tratador funciona como um tratador de arquivo e na chamada são indicados o host, o nome do programa e a versão que será chamada. Também é especificado o protocolo usado. Linha 12 a 15 efetua a chamada do procedimento passando como parâmetros o valor e o tratador do cliente.

No lado do servidor, o procedimento servidor tem que ser escrito. O programa *rpcgen* gera a função *main* para este programa.

```

1  #include    "unpipc.h"
2  #include    "square.h"
3  square_out *
4  squareproc_1_svc(square_in *inp, struct svc_req *rqstp)
5  {
6      static square_out out;
7      out.res1 = inp->arg1 * inp->arg1;
8      return (&out);
9  }

```



A figura acima mostra os passos necessários para a geração dos programas servidor e cliente (server, client), que são:

```
Solaris % rpcgen -C square.x  
Solaris % cc -c client.c -o client.o  
Solaris % cc -c square_clnt.c -o square_clnt.o  
Solaris % cc -c square_xdr.c -o square_xdr.o  
Solaris % cc -o client client.o square_clnt.o square_xdr.o libunpipc.a -lnsl  
  
Solaris % cc -c server.c -o server.o  
Solaris % cc -c square_svc.c -o square_svc.o  
Solaris % cc -o server server.o square_svc.o square_xdr.o libunpipc.a -lnsl
```

A seguir os passos que acontecem em uma chamada remota de procedimento:

O servidor é iniciado e registra-se no mapeador de portas no host do servidor. O cliente também é iniciado e chama a rotina *clnt_create*, que faz contato com o mapeador para encontrar a porta do servidor.

O cliente chama o stub cliente (*squareproc_1*), que prepara mensagem para ser enviada ao servidor.

As mensagens são enviadas pelo cliente o que implica em uma chamada ao kernel (*write*, *sendto*).

Mensagem é transferida pela rede para o sistema remoto (TCP,UDP).

O stub servidor esta esperando mensagens e realiza o procedimento de *unmarshaling*.

O stub servidor realiza uma chamada local (*squareproc_1_svc*) passando os argumentos recebidos.

Quando o servidor termina ele retorna para o stub o resultado.

O stub converte os valores e envia mensagem para o cliente.

A mensagem é transferida pela rede.

O stub cliente recebe mensagem (*read*, *recvfrom*).

O stub finalmente retorna para a função do cliente.