INF01151 – Sistemas Operacionais II

Aula 15 - Remote Procedure Call - RPC

Prof. Alberto Egon Schaeffer Filho

INSTITUTO DE INFORMÁTICA
UFRGS

Introdução

- Necessidade de desenvolver aplicações distribuídas de forma simples (intuitiva, transparente)
  - Solução: estender modelos de programação "convencionais"
  - Utilização de primitivas de mais alto-nível
- Três modelos:
  - **Procedural**: chamada remota de procedimentos (RPC)
  - **Orientado a objetos**: invocação remota de métodos (RMI)
  - **Baseado em eventos**: atendimento de eventos remotos

  Síncrono: Procedural, Orientado a objetos
  Assíncrono: Baseado em eventos
- Provido através de um *middleware* de comunicação

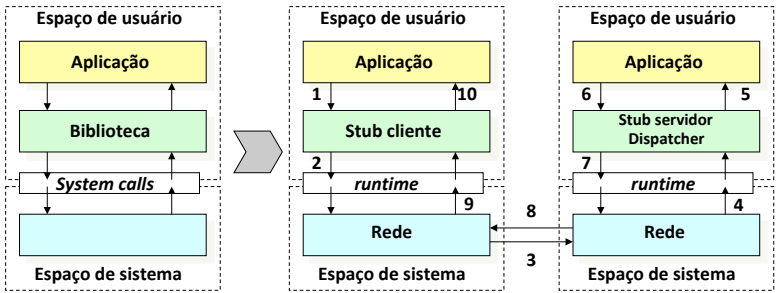
> *"Software que provê um modelo de programação oferecendo abstração de processos, comunicação e serviços através de primitivas próprias"*

Aspectos importantes (ideais) de um *middleware*

- Transparência de acesso e de localização (uniformidade)
  - Mesmo comportamento (sintaxe? e/ou semântica?) independente dos procedimentos serem locais ou remotos
- Protocolo de comunicação (flexibilidade)
  - Comportamento independente do protocolo de transporte (garantias)
- Independência de plataforma (hardware e sistema operacional)
  - Uso de representação externa de dados (*marshalling* e *unmarshalling*)
  - Primitivas para criação de fluxos de execução, sincronização, etc, independentes do sistema operacional
- Uso de linguagens de programação
  - Permitir ou não o emprego de várias linguagens de programação no desenvolvimento da aplicação distribuída

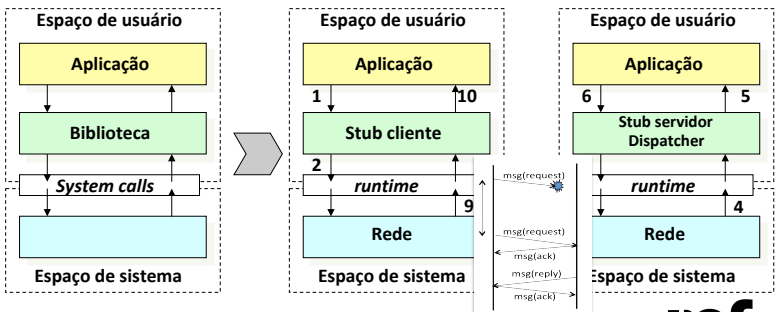
Remote Procedure Call (RPC)

- Generalização do conceito de subrotinas locais
  - Permite um programa chamar um procedimento que não se encontra no espaço de endereçamento do processo



Remote Procedure Call (RPC)

- Generalização do conceito de subrotinas locais
  - Permite um programa chamar um procedimento que não se encontra no espaço de endereçamento do processo



Implementação do mecanismo de RPC

- Oferece transparência sintática através de quatro elementos
  - *Runtime,*
  - *Stub* cliente,
  - *Stub* servidor,
  - *Dispatcher*
- *Middleware* (*runtime*)
  - Responsável pela troca de mensagens entre cliente e servidor
  - Implementa um protocolo **da família request-reply**
    - Necessário definir semântica de invocação, independente da camada de transporte
    - Necessário prover mecanismos para confirmação de recebimentos, retransmissões, time-out, controle de duplicações, históricos, etc
- *Stubs*: implementações "ocas" dos procedimentos
  - Responsáveis por permitir comunicação remota apenas
    - Não implementam "lógica de negócio/serviço"

Stub cliente / Stub servidor

- Programa cliente
  - Stub executa o *marshalling* dos argumentos
  - Envio de uma mensagem (requisição) ao processo servidor
  - Bloqueia até receber mensagem (resposta)
  - Stub efetua *unmarshalling* e repassa à aplicação
- Programa servidor
  - ***Dispatcher*** seleciona qual procedimento stub executar, conforme ID da requisição
  - Stub executa o *unmarshalling* dos argumentos
  - Chama o procedimento associado (chamada local)
  - Stub executa o *marshalling* do resultado
  - Envia mensagem (resposta) para o processo cliente

Geração de stubs:

- **Manualmente**: programador constrói os *stubs* e parâmetros
- **Automaticamente**: baseado em uma *Interface Definition Language* (IDL)
  → Compilador IDL gera código a ser integrado ao do programador

Programando com interfaces

- Maioria das linguagens permite organizar programas como conjuntos de módulos que podem se comunicar
  - Aplicação distribuída = conjunto de módulos que se comunicam entre si
  - Para controlar interações, definir uma **interface** para cada módulo
    - Apresenta apenas serviços que podem ser acessados, esconde-se o resto
    - Implementação dos serviços podem ser alteradas desde que interface permaneça
      - Serviços providos pelo servidor podem ser otimizados sem afetar clientes do módulo
- **Interface** = descrição de procedimentos que podem ser acessados por outros módulos
  - Nome e parâmetros de entrada e saída (inclui tipos)
    - Passagem de parâmetros por **referência** e/ou por **valor**??
  - Passagem de parâmetros é problemática em modelo distribuído
    - E.g.: não faz sentido passar ponteiros
      - Endereços não podem ser passados como argumento ou retornados como resultados de procedimentos

Semântica de passagem de parâmetros

- Passagem **por valor**
  - Parâmetros são copiados na mensagem e enviados ao servidor
  - Não representa maiores problemas
- Passagem **por referência**
  - Sem sentido, em princípio, pois não há espaço de endereçamento comum
  - Soluções **possíveis**:
    - Empregar uma passagem de parâmetros do tipo *call-by-copy-restore*
    - Resolver o ponteiro a cada referência no servidor (mensagens adicionais)
    - Usar mecanismos de DSM (*Distributed Shared Memory*)

Solução na prática: descrever parâmetros na forma entrada/saída

- **Entrada**: enviados ao módulo remoto (mensagem de requisição)
- **Saída**: recebido pelo módulo solicitante (mensagem de resposta)

Interface Definition Language (IDL)

- Interface de serviços
  - Especificação de procedimentos + argumentos (entrada/saída)
    - Modelo RPC
  - Especificação de métodos + argumentos (entrada/saída, referência)
    - Modelo RMI
- Notação para definir interfaces
  - Integrado diretamente em uma linguagem
    - Java RMI
  - Mapeado (compilado) para uma linguagem específica
    - XDR para C, C++

Transparência: Localização de serviços

- *Stub* cliente necessita descobrir o servidor antes de realizar RPC
  - Para quem ele deve enviar a mensagem??
- **Servidor de nomes** [Birrel e Nelson, 1984]
  - Também chamado de *agente de binding*
  - Interface de nomes para serviços remotos
    - **Hostname**: servidor que implementa o serviço
      - Permite a existência de vários servidores
      - Se qualquer servidor pode prover o serviço, então define apenas o tipo (serviço)
    - **Tipo**: especifica a interface (serviço a ser usado)
  - Tipicamente a interface também oferece um **número de versão**
    - Distinção entre versões novas e antigas de um mesmo serviço
    - Manter as antigas durante fase de migração, sistemas legados, compatibilidade, etc

Perguntas:

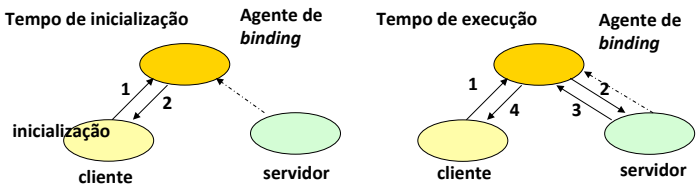
① Como cliente especifica o agente de binding?
② Como o agente de binding localiza o serviço?
③ Quando a localização é realizada?
④ Possível trocar o binding durante execução (on the fly)?
⑤ Pode haver múltiplos bindings?

Agente de binding

- Agente de amarração (*binding*) ou servidor de nomes
  - Permite que o cliente obtenha um **`handler`** para acessar o serviço
    - **Servidores se registram** no agente de binding na sua inicialização
    - **Clientes consultam** o agente de binding para saber quem provê o serviço
  - **Interface do agente de binding**
    - **`register`**: para servidor cadastrar um serviço no agente
    - **`deregister`**: para servidor descadastrar um serviço no agente
    - **`lookup`**: para o cliente localizar um determinado serviço
  - Como localizar um agente de binding?
    - Endereço fixo conhecido e informado a todos clientes e servidores
    - Empregar *broadcasting* para localizar o agente de binding

Quando a localização é realizada?

- Tempo de **compilação**
  - Localização dos serviços e dos servidores é *hard-coded*
- Tempo de **inicialização** do cliente
  - Cliente consulta o agende de binding e armazena resposta em uma cache
    - Chamadas subsequentes não envolvem consultas ao agente de binding
- Tempo de **execução** da chamada
  - Cliente consulta agente de binding na chamada do serviço
    - Agente de binding localiza servidor (possível fazer a chamada em nome do cliente)



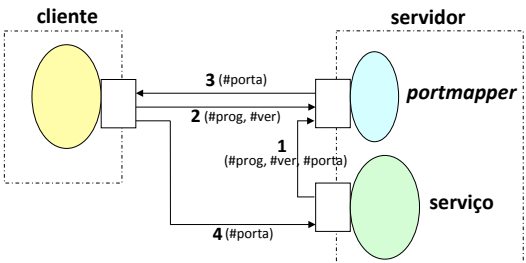
Estudo de caso: *binding* em sistemas Unix

- O agente de *binding* é o daemon **`portmapper`**
  - É um serviço bem-conhecido (porta 111 – Sun RPC)

```
%>rpcinfo –p localhost
```

- Como determinar a máquina servidora?
  - *Hard-coded* ou variável de ambiente
  - *Broadcast* a todos os *portmappers*

```
program vers proto   port
 100000    2   tcp    111 portmapper
 100000    2   udp    111 portmapper
50405925   1   udp  57164
50405925   1   tcp  53652
```



Remote Procedure Call *versus* Local Procedure Call

- Aspectos de aplicações distribuídas que **dificultam a questão da transparência**
  - Falhas parciais
  - Concorrência
  - Latência
  - Acesso a memória

Mais questões sobre transparência

- Em RPC fácil oferecer transparência **sintática**, porém não **semântica**
  - Acesso a **variáveis globais** (como fazer?)
  - Problema com passagem de **parâmetros por referência**
  - Vulnerabilidade a **falhas**
    - Mensagens (*request/reply*) podem ser perdidas
    - Servidor pode pendurar (antes ou depois de ter executado a requisição??)
    - Cliente pode pendurar (antes ou depois de ter aceito o resultado??)
  - A **concorrência é real** (processo local e processo remoto)
  - **Latência da rede** na execução
    - Tempo de execução é sensível a atrasos na rede
- Conclusão: **transparência semântica é difícil** de ser obtida...
  - Pergunta: desejável "esconder" que se trata de comunicação remota?
    - Necessita reagir adequadamente a latência, condições de falhas...
    - Explicitar diferença entre chamada local e remota nas interfaces

Execução de serviços remotos na presença de falhas

- Sistemas distribuídos são mais vulneráveis a falhas
  - Uma vez que envolvem a rede, outro computador e outro processo
- Pontos de falhas por colapso: Cliente? Servidor? Rede?
  - Problema: saber se falha ocorreu antes/depois de processar requisição
  - Impossível distinguir entre falha da rede ou falha do servidor

Falhas no servidor/rede

- Definir uma semântica de invocação
  - No máximo uma vez (***at-most-once***)
  - Pelo menos uma vez (***at-least-once***)
  - Exatamente uma vez (***exactly-once***)
- Implementar operações idempotentes
  - Idempotentes: podem ser realizadas *n* vezes fornecendo o mesmo resultado como tivessem sido executadas uma só vez
    - e.g.: ler um bloco de dados de um arquivo, inserir um elemento em uma posição fixa de um vetor, etc
  - Não idempotentes: o resultado final é modificado pela re-execução
    - e.g.: creditar ou debitar um valor em uma conta bancária, inserir ou remover um elemento no final de uma lista, etc
  - Operações idempotentes simplificam a execução de serviços remotos em caso de falhas

Falhas no cliente

- Cliente “pendura” após ter enviado a requisição
  - Reiniciar o cliente (processo e/ou máquina), porém conseqüência é se ter uma **computação orfã**
- Problemas com computações orfãs:
  - Desperdício de cpu no servidor, *locks* realizados no cliente e não liberados, confusão com a resposta que chega (quem fez?), problemas com números de seqüência, etc
- Soluções possíveis:
  - Extermínio
  - Reincarnação
  - Expiração

Falhas no cliente (continuação)

- Soluções possíveis
  - **Extermínio**:
    - Baseado em logs
    - Cliente loga tudo que executa e em caso de reboot anula todos resquícios de computações orfãs
      - Problema: tamanho do log em disco, e computações disparadas por orfãos
  - **Reincarnação**:
    - Cada reboot gera número (número de época) que é usado para identificar request e reply
      - Anula tudo que pertence a uma época anterior a atual
  - **Expiração**:
    - Cliente deve confirmar a recepção do resultado (commit)
      - Se servidor não receber commit, desfazer operação (transações)

Estudo de caso: Sun RPC

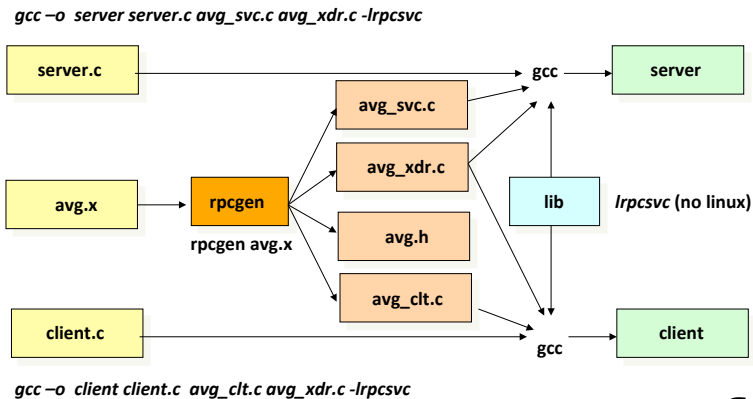
- Projetado para cliente/servidor no Sun NFS (Network File System)
  - Semântica adotada é ***at-least-once*** (ações no NFS são idempotentes)
    - Se usado em outro contexto, cabe ao usuário fazer tratamento adequado
  - Sun RPC é parte de **Sun OS** e outros sistemas operacionais **UNIX**
  - Implementações tem escolha de implementar RPC sobre **UDP ou TCP**
- Define uma linguagem (**`XDR`**) e compilador (**`rpcgen`**) para IDLs
- Características gerais da XDR:
  - Originalmente projetada p/ especificar representação externa de dados
  - Especifica definições de procedimentos, parâmetro in/ parâmetro out
    - Uso de structs para permitir mais campos
  - Não fornece servidor de nome global
    - Usa **`portmapper`**: servidor de nome local (porta x interface,versão)
      - Cliente deve conhecer a máquina que executa o serviço (endereço IP ou nome DNS simbólico)

Program number
Version number

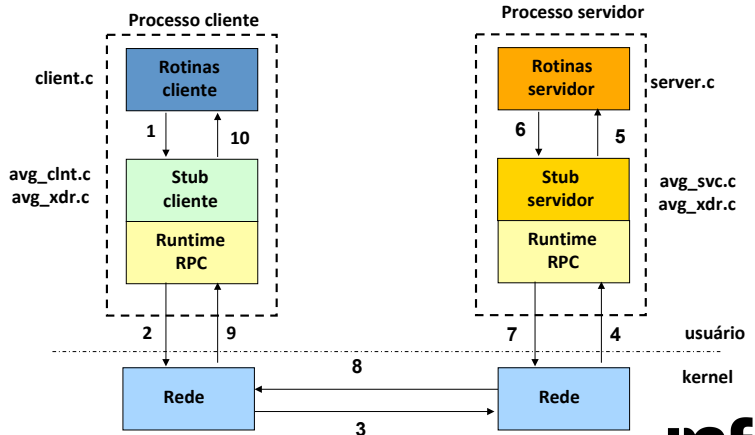
Desenvolvimento em Sun RPC

- Como usar um procedimento remoto? Passos:
  1. Escrever uma definição de interface usando XDR
     - Notação primitiva, sintaxe baseada em C
     - Contém a especificação do procedimento remoto e parâmetro
  2. Compilar com **`rpcgen`** gerando:
     - Stub cliente
     - Stub servidor
     - Arquivos .h
     - Duas funções de manipulação de parâmetros a serem usadas pelo cliente e pelo servidor (fazem o *marshalling* e o *unmarshalling*)
  3. Programa cliente é escrito, compilado e ligado junto com o stub cliente
     - usa o arquivo .h
  4. Programa servidor é escrito, compilado e ligado junto com o stub servidor
     - usa o arquivo .h

Exemplo: `rpcgen` (*Sun* RPC IDL)



Passos envolvidos na execução



Leituras adicionais

- Couloris, G.; Dollimore, J.; Kindberg, T. and Blair, G.– *"Distributed Systems: Concepts and Design*" (5th edition), Addison Wesley, 2012
  - Capítulos 5 (seção 5.3)
- Remote Procedure Calls – Linux Journal (1997)
  - http://www.linuxjournal.com/article/2204