

# Introdução à passagem de mensagens e sistemas distribuídos

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos  
Departamento de Informática  
Universidade do Minho

2007/2008



# Mensagens versus memória partilhada

- Duas grandes famílias de paradigmas.
- Memória partilhada + primitivas de sincronização:
  - semáforos,
  - monitores,
  - ...
- Memória local + passagem de mensagens:
  - formalismos de concorrência: CSP, CCS, ...
  - paradigmas para programação distribuída: cliente-servidor, invocação remota, objectos distribuídos, ...



# Memória local + passagem de mensagens

- Cada processo apenas acede a memória sua;
- Não há variáveis partilhadas;
- Mensagens único meio de:
  - comunicação entre processos,
  - sincronização entre processos.
- Modelo natural para sistemas distribuídos.

## Sistema Distribuído

Conjunto de computadores ligados em rede, com software que permita a partilha de recursos e a coordenação de actividades, oferecendo idealmente um ambiente integrado.



# Aspectos de sistemas de mensagens

Alguns aspectos a ter em conta:

- Primitivas síncronas versus assíncronas;
- Endereçamento e identidade;
- Orientação à conexão versus datagramas;
- Unicast, multicast e broadcast;
- Passagem de dados versus dados+código.



# Primitivas síncronas versus assíncronas

- Dois processos: um envia mensagem, outro recebe.
- Duas primitivas: send e receive;
- O receive tem que bloquear até mensagem ser recebida.
- No send há duas variantes:
  - assíncrona: emissor não bloqueia; prossegue independentemente;
  - síncrona: emissor bloqueia até ser feito receive;
- Uma variante pode ser obtida à custa da outra:
  - assíncrona pode ser obtida à custa de buferização por intermediário;
  - síncrona pode ser obtida com uma mensagem de resposta, à custa de um processo extra no receptor;
- Variante síncrona é mais natural ponto-a-ponto; assíncrona quando há várias etapas pelo meio.



# Endereçamento e identidade

- Endereçamento é assimétrico:
  - emissor necessita de saber para onde mandar mensagem;
  - receptor não necessita saber de onde veio; mas pode ser informado se útil.
- Como identificar o receptor de uma mensagem?
- Duas hipóteses gerais:
  - identidade do processo receptor;
  - entidade independente; e.g canal, porto.
- Um canal permite que:
  - um ou mais processos enviem mensagens para o canal;
  - para serem recebidas por um de vários processos do outro lado.
- Identidade como entidade de primeira classe:
  - é importante a identidade poder ser passada em mensagens.



# Orientação à conexão versus datagramas

- Por cima de um protocolo de rede, como IP, são oferecidos dois tipos de protocolos de transporte:
  - orientados à conexão, como TCP;
  - orientados ao datagrama, como UDP;
- Com a orientação à conexão:
  - há um custo inicial de estabelecimento; e.g. 3-way handshake;
  - o uso subsequente da conexão é rentabilizado;
  - a conexão é vista pela aplicação como um *stream*, para onde pode ser enviado e recebido um fluxo contínuo de dados;
  - o protocolo de transporte encarrega-se de recuperar erros e controlar fluxo, transparentemente à aplicação.
- Com a orientação ao datagrama:
  - são enviadas mensagens individuais de tamanho curto;
  - mensagens podem ser perdidas, chegar fora de ordem ou duplicadas;
  - aplicação tem que recuperar de erros; mais complexo;
  - mais baixo nível, usado e.g em jogos multi-utilizador.



# Unicast, multicast e broadcast

- Quantos receptores pode ter uma mensagem?
- Três casos são significativos:
  - unicast: existe apenas um receptor;
  - multicast: pode ser especificado um conjunto de receptores;
  - broadcast: mensagem é enviada para todos “perto”.
- Algoritmos que usem multicast ou broadcast podem ser eficientes havendo suporte de rede para estas primitivas.
- Com suporte de rede, um broadcast é efectuado gastando o mesmo tempo que um unicast; poupa-se o envio sequencial de uma série de mensagens individuais.
- Exemplos:
  - broadcast para um segmento de uma LAN;
  - broadcast para vizinhos (com proximidade geográfica) numa rede de sensores;



# Passagem de dados versus código

- A passagem de estruturas de dados necessitam de ter em conta:
  - heterogeneidade de hardware;
  - heterogeneidade de linguagens;
- Mobilidade de código é mais complicado de suportar:
  - facilitado por máquinas virtuais;
  - permite padrões de interacção mais sofisticados;



# História

- Inicialmente sistemas centralizados multi-utilizador.
- Sistemas Distribuídos originados com o aparecimento de LAN's (1970's) interligando workstations.
- Fomentados por PC's de baixo custo e alto desempenho.
- Evolução desde LAN's até à WAN global que é a Internet.



# Evolução da exploração de sistemas distribuídos

- Sessão remota (telnet) e transferência explícita de ficheiros.
- Sistemas de ficheiros distribuídos.
- Generalização do paradigma cliente/servidor e RPC.
- Sistemas de objectos distribuídos: uso de middleware como JAVA RMI ou CORBA.
- A World Wide Web; aumento da importância do HTTP e formatos baseados em texto, com o sucesso da WWW.
- Jogos em rede; ultimamente aparecem os MMOGs: *massively multiplayer online games*.
- Popularização de sistemas peer-to-peer.



## Exemplo: Unix distribuído

- Fornecer modelo do Unix com melhores performances e facilidades do que em computadores multi-utilizador.
- Computadores com poder de processamento para cada utilizador, em vez de acesso por terminal em time-sharing a computador central.
- Servidores para acesso a recursos partilhados como impressoras e disco.
- Motivou o aparecimento de ferramentas para programação como o RPC (Remote Procedure Call), utilizadas para implementar abstrações como o NFS (Network File System).



# Exemplo: WANs

- Crescimento exponencial da Internet.
- Software tem de ser projectado tendo em conta a escalabilidade.
- Exemplos:
  - resolução de nomes (DNS),
  - correio electrónico,
  - USENET news,
  - WWW.



# Exemplo: aplicações comerciais

- Aplicações “sérias” como:
  - Reserva de bilhetes em companhias de aviação.
  - Comércio electrónico.
  - Máquinas Multibanco.
  - Infra-estrutura de acesso a uma bolsa de valores.
- Exigem fiabilidade e segurança, mesmo em presença de falhas de comunicação e de computadores.



# Exemplo: aplicações interactivas e multimédia

- Dois aspectos ortogonais: largura de banda e latência.
- Aplicações podem ser exigentes em termos de largura de banda.  
*Exemplo: video-on-demand.*
- Aplicações interactivas, mesmo com poucas necessidades em largura de banda, podem ser exigentes em termos de latência.  
*Exemplo: jogo multi-utilizador do tipo first-person-shooter.*
- TCP versus UDP.
- Jogos multi-utilizador usam tipicamente UDP para comunicação.



# Caracterização dos sistemas distribuídos

- Elementos autónomos de processamento
- Elementos autónomos de armazenamento
- Comunicação via mensagens
- Falhas independentes
- Assincronia
- Heterogeneidade



# Elementos autónomos de processamento

- Os elementos são autónomos; não existe um único ponto de controlo.
- A concorrência é uma característica fundamental:
  - utilizadores do sistema podem fazer pedidos concorrentemente,
  - elementos podem responder concorrentemente a pedidos que lhes cheguem,
  - elementos podem colaborar concorrentemente na resolução de pedidos.
- Tempos de vida das entidades podem variar muito.



# Elementos autónomos de armazenamento

- Cada elemento pode armazenar localmente informação.
- O acesso local é mais eficiente . . .
- o que motiva *caching* e replicação de informação . . .
- o que levanta problemas de coerência entre réplicas; e.g. se forem permitidas escritas concorrentes não controladas;
- e motiva mecanismos de etiquetamento e controlo de versões.
- Temos coerência forte se o conjunto de réplicas puder ser visto como equivalente a uma única instância (*one-copy equivalent*).
- Existe um compromisso entre coerência e disponibilidade.



# Comunicação via mensagens

- Num programa clássico existem variáveis globais partilhadas.
- Num sistema distribuído não existem dados globais partilhados.
- Toda a comunicação tem que ser *fisicamente* via mensagens.
- A programação de sistemas distribuídos pode ser explicitamente via mensagens ...
- ou podem ser utilizadas abstrações de mais alto nível que escondam a troca de mensagens; e.g. invocação remota.



# Falhas independentes

- Num sistema distribuído as falhas podem ser em:
  - elementos de processamento,
  - canais de comunicação: perda, duplicação ou reordenação de mensagens.
- Num sistema distribuído, uma falha num elemento pode não impedir os restantes de prosseguirem.
- Se não vem a resposta a um pedido, o que quer isso dizer?
  - houve falha de rede?
  - houve falha no nó destino do pedido?
  - como as distinguir?
  - como proceder em seguida?
- Falhas e assincronia são a maior fonte de complexidade e exigem paradigmas e algoritmos sofisticados para as atacar.



# Assincronia

- Num sistema distribuído podem ser feitas diferentes suposições relativamente à sincronia, dependendo do caso: e.g. LAN versus WAN.
- Num sistema completamente assíncrono temos:
  - desconhecimento das velocidades relativas de processamento,
  - inexistência de um limite para o tempo de comunicação,
  - inexistência de um relógio global.
- Cada nó tem conhecimento local que vai propagando por troca de mensagens.
- Foram desenvolvidos mecanismos de etiquetamento lógico que não usam o tempo físico, como os relógios vectoriais.



# Heterogeneidade

- Um sistema distribuído evolui com o tempo: são acrescentadas novos componentes e são removidos outros;
- O sistema pode ser programado em diferentes linguagens:  
*e.g. clientes em Java e Python, servidor em C++.*
- Diferente hardware pode ser utilizado, levando a diferentes:
  - representações de dados,
  - código máquina.
- Diferente *middleware* pode ser utilizado, como JAVA RMI ou CORBA, levando à necessidade de *bridges*.



# The Eight Fallacies of Distributed Computing

“Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.”

Peter Deutsch

- 1 The network is reliable
- 2 Latency is zero
- 3 Bandwidth is infinite
- 4 The network is secure
- 5 Topology doesn't change
- 6 There is one administrator
- 7 Transport cost is zero
- 8 The network is homogeneous



# Paradigmas de programação de sistemas distribuídos

Diferentes paradigmas:

- Paradigma cliente-servidor.
- Invocação remota de procedimentos; exemplo: Sun RPC.
- Invocação de operações em sistemas de objectos distribuídos; exemplos: Java RMI, CORBA.
- Abstracções de comunicação de mais alto nível, oferecendo diferentes garantias; e.g. comunicação em grupo.



# Paradigma Cliente-servidor

- Paradigma clássico em sistemas distribuídos.
- Clientes faz pedidos.
- Servidor processa pedidos e envia respostas.
- Modelo muito genérico.
- A maioria das aplicações distribuídas baseiam-se em cliente servidor:
  - WWW;
  - Mail;
  - NFS (Network File System)
  - Jogos tipo FPS;
  - ...



# Invocação remota de procedimentos — RPC

- O desenvolvimento de aplicações pelo paradigma cliente/servidor pode ser efectuado com diferentes ferramentas.
- A utilização directa de primitivas básicas tipo send/receive (como sockets) revela-se de demasiado baixo nível.
- Uma RPC (Remote Procedure Call) aproxima a programação de uma aplicação distribuída à programação tradicional, em linguagens procedimentais.
- Tal é conseguido fazendo a invocação de uma operação num servidor remoto parecer no código cliente como uma simples invocação local de um procedimento.



# Sistemas de objectos distribuídos

- Objectos e invocação remota apropriados para distribuição.
- Utilização de linguagens OO: um serviço distribuído é oferecido a um cliente como um objecto com uma dada interface.
- Acesso local e encapsulamento: implementação de um serviço acede a estado local ou a outros serviços via invocação remota.
- Referências para um serviço podem ser passadas e devolvidas: identidade de serviços como entidade de primeira classe.
- Serviços distribuídos genéricos (nomes, eventos, concorrência, transacções, . . . ) são oferecidos com interfaces standardizadas.



# Sistemas de notificação de eventos

- Permite obter desacoplamento na comunicação entre objectos.
- Modelo *publish-subscribe*.
- Publicadores geram eventos que anunciam ao serviço.
- Subscritores anunciam em que tipo de eventos estão interessados.
- Publicadores e subscritores não se conhecem mutuamente.
- Serviço de eventos trata de encaminhar os eventos.
- Permitem evitar *polling* e transmissões ponto-a-ponto redundantes; são em muitos casos a única solução escalável.



# Comunicação em grupo

- Para implementar certos conceitos, como replicação, pode ser necessário enviar mensagens a vários processos (as réplicas).
- Pode ser necessário que, por exemplo, as mensagens cheguem a todos, ou por uma dada ordem.
- A implementação via primitivas simples é complexo sobretudo tendo em conta falhas de transmissão (perda, duplicação ou troca de ordem de chegada) ou falha de processos.
- Comunicação em grupo é um paradigma de mais alto nível para comunicação de mensagens a um grupo de processos.
- Diferentes garantias podem ser oferecidas — atomicidade, ordem — com diferentes custos de implementação.



# Algoritmos distribuídos

- Para evitar um ponto de falha único, um modelo apropriado será ter  $N$  entidades independentes em que nenhuma é “especial”.
- Contrariamente ao modelo cliente-servidor, nos algoritmos distribuídos  $N$  intervenientes comportam-se tipicamente de um modo simétrico, executando todos o mesmo algoritmo.
- Estes algoritmos consistem na troca de mensagens entre os intervenientes com o fim de resolver um dado problema.
- Muitos deles têm o conceito de ronda; durante uma ronda um dado processo pode ser especial (coordenador), mas não indispensável: se houver uma falha o algoritmo pode iniciar uma nova ronda com um novo coordenador.
- Estes algoritmos são frequentemente para sistemas assíncronos, não fazendo uso de tempo: conceitos como rondas fazem uso de contadores.

