Introdução ao Erlang

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Concorrência cada vez mais importante

- Vulgarização do paralelismo:
  - sistemas SMP, clusters
  - vários "cores"por processador;
- Software como serviço:
  - mais aplicações remotas;
  - necessidade de servir muitos clientes;
- Pub/Sub, Message Queues, Instant Messaging:
  - AMQP (Advanced Message Queuing Protocol)
  - XMPP (jabber)
- Web 2.0:
  - aplicações mais dinâmicas, AJAX;
  - Comet-style applications: event-driven server-push data-streaming, envolvendo conexões abertas para possivelmente milhares de clientes.

Paradigma de concorrência mais vulgarizado

- O paradigma em uso nas linguagens mais populares (C, C++, Java, C#, ...) é baseado em memória partilhada;
- Primitivas clássicas:
  - semáforos (Dijkstra),
  - monitores (Brinch Hansen / Tony Hoare)
- Actualmente o normal é:
  - threads (memória partilhada) +
  - monitores;
- Sistemas baseados em mensagens (CSP, CCS, Actores) são normalmente vistos como “coisas académicas”.

Problemas com memória partilhada + monitores (I)

- Código de controlo de concorrência difícil de programar;
- broadcast ingénuo; optimizações trazem bugs subtis;
- A variante actual de monitores é uma combinação explosiva:
  - ausência de memória do signal;
  - ultrapassagens: estado pode mudar arbitrariamente entre o signal e o wait;
  - spurious wakeups;
- Não escala nem se adapta naturalmente a sistemas distribuídos;

Problemas com memória partilhada + monitores (II)

- É difícil programar sistemas complexos envolvendo milhares de entidades partilhadas;
- Exemplo: jogos multi-jogador;
- Segundo Tim Sweeney (Epic Games, Unreal Engine) é típico:
  - 30–60 updates por segundo;
  - 10000 objectos activos;
  - cada objecto quando actualizado afecta 5 a 10 outros;

  ”Manual synchronization (shared state concurrency) is hopelessly intractable here”

Problemas com memória partilhada + monitores (III)

- Se é difícil programar para o caso normal . . .
- . . . é um pesadelo quando as coisas correm mal;
- Um erro numa thread não é isolado e afecta todo o sistema:
  - e.g. uma thread gera uma excepção sem libertar lock;
  - difícil de tratar no caso geral com muitos locks envolvidos;
- Se já é desagradável com apenas 1 lock envolvido . . .

```
public void m() {
  lock.lock();  // block until condition holds
  try {
    // ... method body
  } finally {
    lock.unlock()
  }
}
```

Problemas com memória partilhada + programação imperativa

- É difícil “voltar para trás” quando as coisas correm mal;
  - uma thread quer executar acções A e B;
  - repara que não consegue executar B;
  - quer desfazer A, mas efeito já foi para memória partilhada, e possivelmente utilizado por outras threads;
- Problema também do paradigma imperativo;
- Conceitos como STM (software transactional memory) poderão ajudar, se viáveis; ainda incógnita.

What will a definitive programming language look like?

Convergence in Language Design: A Case of Lightning Striking Four Times in the Same Place (Peter Van Roy);

- Apresenta quatro casos de linguagens/sistemas criados para fins mais diversos como:
  - tolerancia a faltas,
  - programação distribuída segura,
  - programação distribuída com transparência de rede,
  - ensino de programação como disciplina unificada;
- Padrão recorrente – estrutura por camadas:
  - núcleo puramente funcional;
  - concorrência deterministica (declarativa/dataflow);
  - concorrência por passagem assincrona de mensagens;
  - estado partilhado global, normalmente transacional;
- Um deles é o Erlang/OTP.

Erlang

- Criada pela Ericsson; com o nome do matemático A. K. Erlang
- Linguagem dinamicamente tipada;
- Núcleo puramente funcional estrito;
- Processos ultra-leves geridos pela máquina virtual (não SO);
- Concorrência por mensagens assincronas (modelo de actores);
- Base de dados persistente e transaccional para estado global;
- privilegia fiabilidade; sistemas always-on;
- privilegia previsibilidade de resposta; sistemas soft realtime.

Erlang: linguagens funcionais até podem ser usadas “a sério”

- Apesar de simple e elegante, Erlang não é uma toy-language;
- É das linguagens funcionais com mais sucesso:

AXD 301: switch ATM da Ericsson; 1.7 Mlocs Erlang; 99.999999999 % fiabilidade (31ms/ano);

GPRS: 76% código em Erlang;

ejabberd: servidor Jabber/XMPP distribuído, adoptado por jabber.org; 250000 utilizadores simultâneos num domínio; facilmente 10000 utilizadores por máquina;

Yaws: servidor web; funciona bem com 80000 conexões num teste em que o Apache morreu às 4000;

RabbitMQ: implementação do standard de messaging AMQP (Advanced Message Queuing Protocol);

Erlang: núcleo funcional

- Linguagem funcional:
  - funções de ordem superior,
  - pattern matching,
  - sem atribuição (single assignment),
  - estruturas de dados principais: tuplos e listas;
- Linguagem dinamicamente tipada:
  - “linguagem de scripting” funcional,
  - muito boa para prototipagem rápida,
  - ferramentas de verificação estática; e.g. dialyzer;

Processos Erlang

- Processo Erlang é uma entidade gerida pela máquina virtual;
- Muito pouco custo de criação;
- Podemos ter facilmente centenas de milhares de processos Erlang;
- Muito pouco custo de comutação (uma ou duas ordens de magnitude mais rápido do que threads do SO);
- Algumas threads do SO são usadas pela máquina virtual, para explorar paralelismo ou I/O;
- Cada processo pode manter um estado local, e comunicar por mensagens com outros processos.

Envio e recepção de mensagens

- Erlang segue o modelo de actores;
- Cada processo tem uma mailbox – fila de mensagens;
- mensagens são enviadas especificando o processo de destino – não é usado o conceito de canal.
- Envio é assincrono; não devolve erro mesmo que o processo destino já tenha terminado;
- Mensagens são armazenadas na mailbox pela ordem de chegada;
- Uma mensagem é removida da mailbox com uma operação bloqueante via pattern matching.

Gestão de memória

- Linguagem funcional com garbage collection (GC);
- Processos não partilham memória;
- Numa linguagem funcional, com estruturas imutáveis, a ausência de partilha poderia ser apenas conceptual . . .
- . . . mas é física:
  - cada processo tem o seu próprio stack e heap;
  - mensagens envolvem cópia de estruturas de dados;
- Porque?
  - mensagens são tipicamente curtas;
  - heap por processo leva a GC por processo, simples;
  - às vezes sistemas são afinados para certos processos serem criados sem nunca provocar GC durante o seu tempo de vida;
  - evita que GC cause uma pausa grande no sistema;
- Há uma opção de heap partilhada que, em geral, tem pior comportamento e não é normalmente usada.

Processos Erlang como unidade de encapsulamento

- Encapsulamento de estado em linguagens imperativas / objectos é um dos grandes problemas ainda não resolvidos;

  "The big lie of object-oriented programming is that objects provide encapsulation" — *John Hogg*

- Mistura de estado mutável com estilo funcional (e.g. em linguagens como Python) traz surpresas (bugs) curiosos;
- Em Erlang, um processo encapsula realmente estado;
- Pode ser visto como um objecto:
  - utilizado por clientes via funções de interface,
  - estas encapsulam passagem de mensagens;

Previsibilidade e rapidez de resposta

- Erlang privilegia respostas rápidas a eventos externos;
- Usado em sistemas soft realtime;
- Reflectido no desenho da linguagem / sistema:
  - não introduzir conceitos que tornem difícil estimar desempenho; e.g. não tem lazy evaluation;
  - processos ultra-leves com comutação de contexto muito rápida;
  - GC por processo para evitar pausas longas;
  - base de dados em memória (Mnesia); isolamento

Fiabilidade sistémica

- Em sistemas grandes, bugs são inevitáveis;
- Sistema deve funcionar, mesmo com partes incorrectas;
- Filosofia Erlang – evitar programação defensiva:
  - funções não se protegerem contra argumentos inválidos;
  - não espalhar e misturar código de verificação de erros com o código que cobre o caso normal;
- Filosofia Erlang – fail-crash:
  - se função não consegue fazer o suposto não tenta devolver erro ao contexto invocador, mas aborta o processo;
  - exemplo: se guardas de um “if” ou os padrões de um “case” não cobrem todos os casos, o processo aborta;
- Processos essenciais para o isolamento e contenção de erros:
  - processos podem ser ligados entre si;
  - processos supervisores podem relançar processos que falharam;
  - sistema como árvore de supervisão;

Operação non-stop e hot code swapping

- Erlang usado em sistemas que não podem ser desligados;
- Com o tempo há a necessidade de novas versões de código:
  - um bug provocou um crash num processo, sendo gerado um *error report*; depois o bug é corrigido;
  - nova funcionalidade é acrescentada à aplicação;
- Erlang permite carregar uma nova versão de um módulo numa aplicação a correr;
- Duas formas de invocar função:
  - intra-módulo; usa a mesma versão do módulo;
  - com prefixo do módulo: usa a versão mais recente;
- VM suporta coexistência de duas versões de um módulo;
- Diferentes processos podem correr diferentes versões;

Estado global – ets e mnesia

- A última camada permite estado visível por todos os processos;
- Dois componentes pricipais: ets e mnesia;
- ets – Erlang Term Storage:
  - Permite criar tabelas com muitas entradas;
  - Em memória, via tabelas de hash ou árvores balanceadas;
  - Pertencem ao processo criador (e por omissão são removidas quando o processo termina);
  - Podem ser criadas como private, protected, ou public, afectando o uso por outros processos;
  - Guardada em área de memória própria, sem GC;
  - Termos são cópiados entre processos e tabelas;

Estado global – mnesia

"Mnesia is a distributed DataBase Management System (DBMS), appropriate for telecommunications applications and other Erlang applications which require continuous operation and exhibit soft real-time properties."

- Escrita em Erlang, fazendo uso do ets;
- Permite armazenar termos genéricos, sem impedância;
- Query Language baseada em compreensões de listas:
- Persistência; mas normalmente DB em memória;
- Transacções aninhadas usando simples funções Erlang;
- Replicação e fragmentação de tabelas;
- Controlo de nível de isolamento;
- Acessos de leitura extremamente rápidos se necessário;
- Permite reconfiguração sem parar o sistema;

Ausência de impedância – de volta à simplicidade

- Aplicações actuais tendem a ser construídas usando uma miríade de componentes que têm de ser combinadas;
- A impedância entre estas complica o desenvolvimento;
- Erlang + Mnesia + Yaws permite o desenvolvimento num ambiente integrado de aplicações escaláveis;
- Novo acrónimo – LYME: Linux, YAWS, Mnesia, Erlang;

"Erlang + Yaws is so much easier to setup and install than Ruby + Gems + Rails + FastCGI + Lighttpd + MySQL + MySQL bindings + every-other-package-in-the-universe. I've been slaving away at these installation instructions for OS X for what feels like an eternity, and my patience is running very low." *Yariv Sadan*

Sistema de runtime e ferramentas

Erlang/OTP possui bastantes ferramentas maduras; exemplos:

Dialyzer: ferramenta de análise estática para detecção de erros (tipos, código não atingível, testes desnecessários); faz análise de fluxo de dados para determinar "disjoint union of prime types";

Application Monitor: ferramenta gráfica que permite monitorizar a árvore de supervisão de aplicações (locais ou remotas);

Event Tracer: permite recolher e vizualizar informação usando o mecanismo de tracing;

Process Manager: permite inspeccionar processos (locais ou remotos) e fazer tracing;

Table Visualizer: permite inspeccionar e modificar tabelas ets ou mnesia;