Sistemas Distribuídos com Redes de Sensores sem Fio

INF2545

Sistemas Distribuídos

- o que são: "coleção de máquinas independentes que aparecem para o usuário como um único sistema coerente"
  - que tipo de usuário?
    - programador é usuário?
  - o que é "coerente"?
    - o conceito de transparência
- um sistema distribuído é uma coleção de máquinas independentes usadas em conjunto para executar uma tarefa ou prover um serviço.

PUC RIO

Received: by jumbo.dec.com (5.54.3/4.7.34)
id AA09105; Thu, 28 May 87 12:23:29 PDT
Date: Thu, 28 May 87 12:23:29 PDT
From: lamport (Leslie Lamport)
To: src-t
Subject: distribution
There has been considerable debate over the years about what constitutes a distributed system. It would appear that the following definition has been adopted at SRC:

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

The current electrical problem in the machine room is not the culprit--it just highlights a situation that has been getting progressively worse. It seems that each new version of the nub makes my FF more dependent upon programs that run elsewhere.
Having to wait a few seconds for a program to be swapped in is a lot less annoying than having to wait an hour or two for someone to reboot the servers. ...
I will begin the effort by volunteering to gather some data on the problem. If you know of any instance of user's FF becoming inoperative through no fault of its own, please send me a message indicating the user, the time, and the cause (if known).
Leslie

para que queremos SDs?

- custo e desempenho
  - paralelismo e computação distribuída
- escalabilidade
  - facilidade de aumentar recursos
- distribuição inerente
  - dispersão geográfica e social
  - compartilhamento de recursos
- confiabilidade
  - redundância

PUC RIO

mas...

- rede tem que ser levada em consideração
  - desempenho e falhas
- redundância introduz a necessidade de coordenação
- complexidade de software
- segurança

  - distribuição introduz problemas inexistentes em sistemas centralizados

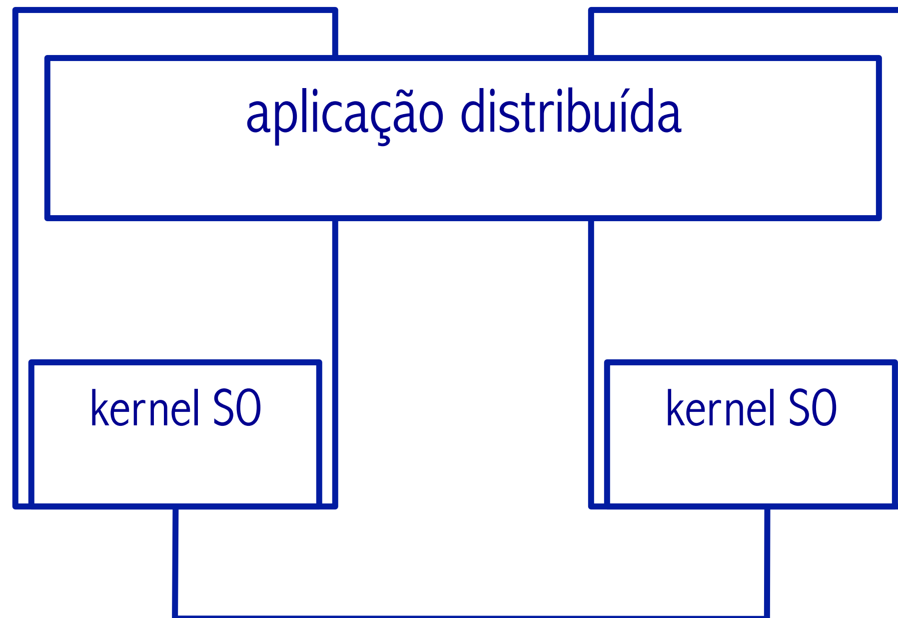
aplicações distribuídas

- comunicação
- colaboração
  - email
  - jogos
  - distribuição de conteúdo
- acesso uniforme a recursos
  - single system image
  - nuvens?
  - sistemas de arquivos distribuídos
- monitoramento
  - sensores

PUC RIO

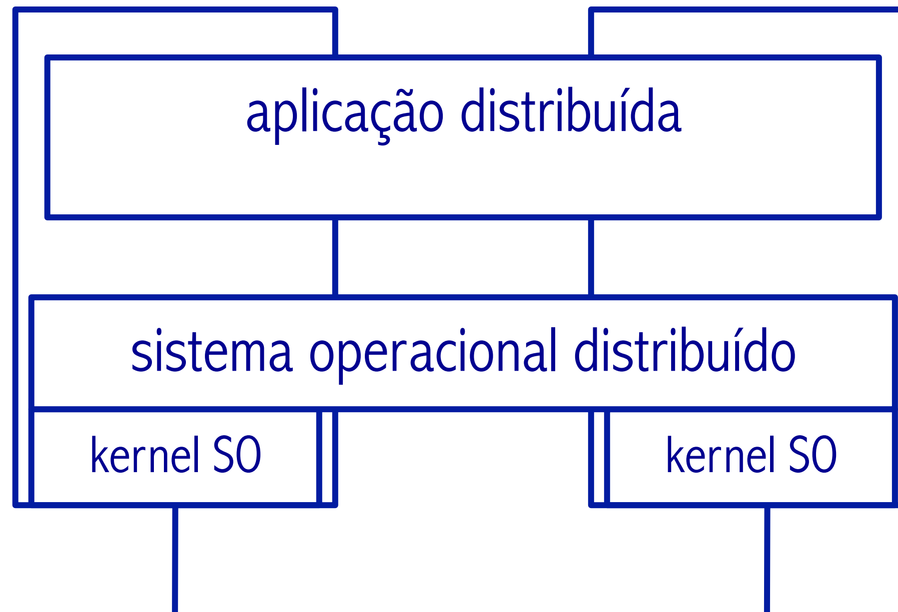
arquiteturas de interesse

- multicomputadores: cada um com sua memória e processador(es?), interligados por redes
  - construção de aplicação distribuída sobre recursos oferecidos por SOs pode ser árdua



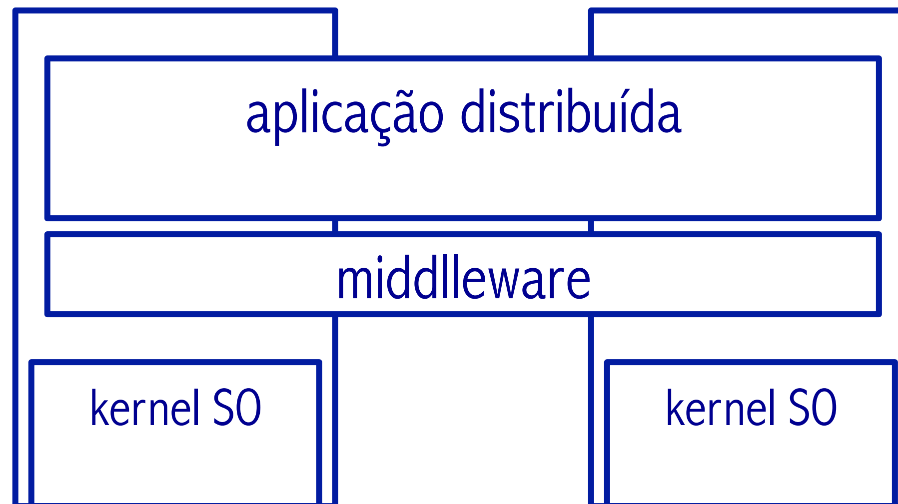
SOs distribuídos

- interesse especial nos anos 80
- sistemas que controlariam o conjunto de recursos de várias máquinas
  - remanescentes importantes:
    - servidores de arquivos, técnicas de segurança, ...



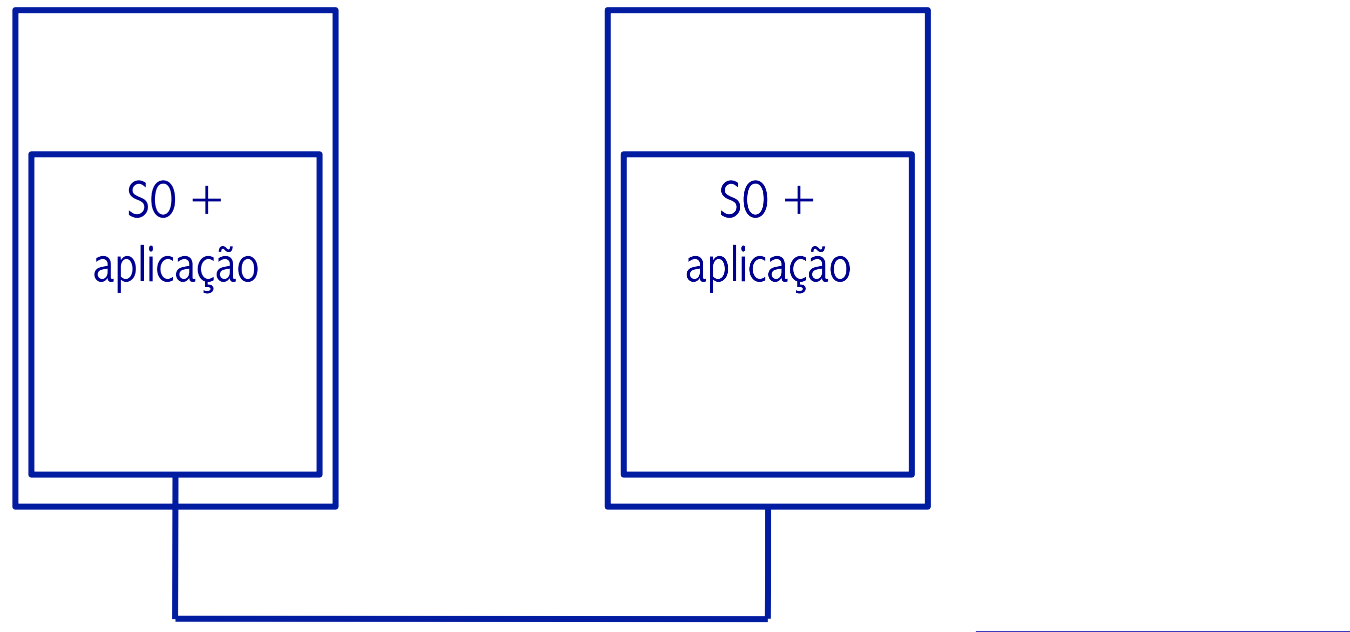
Middleware

- serviços e abstrações que facilitam o desenvolvimento de aplicações distribuídas



sistemas limitados

- redes de sensores são multicomputadores?
  - construção de aplicação distribuída sobre recursos oferecidos por SOs com MUITAS limitações de recursos
  - SOs tipicamente são bibliotecas oferecidas para ligação com aplicação



acoplamento

- aplicações/arquiteturas classificadas em *forte ou fracamente acopladas* de acordo com o grau de interdependência entre as partes
  - aplicações científicas
  - correio eletrônico
  - jogos
  - ...
    - relação com escalabilidade

PUC RIO

estudo de SDs

- livros clássicos de sistemas distribuídos
  - comunicação entre processos
  - sincronização
  - sistemas de arquivos
  - segurança
  - confiabilidade

  ...em geral como assuntos estanques
- nesse curso, ênfase na programação de SDs e na interrelação entre comunicação, sincronização, confiabilidade e escalabilidade
  - uso do cenário de redes de sensores para concretizar os problemas discutidos
    - motes praticamente não fazem sentido sem comunicação
    - padrões de interação diferentes conforme natureza da aplicação
  - programação em ambientes mais convencionais: Lua

PUC RIO

aplicações em rssf

- coleta de informação para análise offline ou decisões extenas
    - periódica, alarmes etc
    - agregação/estatísticas
- controle
    - sensores e atuadores(? actuators)
- características
    - limitação de recursos
    - programming in the small X programming the network
    - modelo de carga de programas: dificuldade de recolher equipamento
    - falhas, falhas, falhas, ...

programa do curso

- Cenários de execução distribuída. Aplicações fraca e fortemente acopladas. Escalabilidade
- Processos. Modelos de concorrência. Threads. Eventos.
- Comunicação. Troca de mensagens. Sincronismos e assincronismo. Modelos de Comunicação.
- Arquiteturas. Cliente-servidor. Peer-to-peer. Publish/subscribe.
- Segurança. Ameaças e técnicas. Autenticação e autorização.
- Tolerância a falhas. Redundância. Replicação.
- Coordenação. Ordenação de mensagens. Exclusão mútua. Eleição de lider.
- RSSF. Programação de dispositivos com recursos limitados. Ambiente de desenvolvimento tinyOS/nesC.

PUC RIO

questões

- desempenho
- transparência
- escalabilidade
- flexibilidade
- facilidade de desenvolvimento
- confiabilidade

PUC RIO

Avaliação

- trabalhos no laboratório + 3 entregas
- 3 resumos ou críticas de artigos – individuais
- 1 trabalho final (implementação, texto e apresentação)

- sobre prazos de entrega:
  - Cada aluno terá direito a 8 dias de atraso por semestre. Depois de gastos esses dias, trabs serão desconsiderados.

PUC RIO

Bibliografia

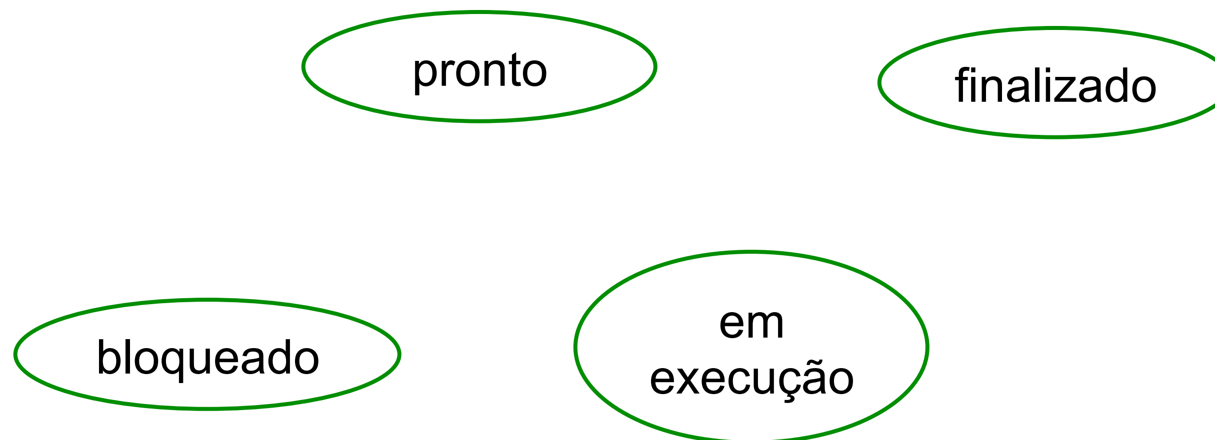
- A. Tanenbaum e M. van Steen. Distributed Systems: Principles and Paradigms. Prentice-Hall, 2007.
- K. Birman. Reliable Distributed Systems. Springer, 2005.
- surveys e artigos

introdução

- processos e concorrência

Processos e Concorrência

- processo: modelo de execução sequencial
  - "programa em execução"
  - código, dados globais, pilha de execução
  - estrutura presente em qualquer sistema operacional
    - escalonamento
    - diferença para processos de livros de SO



concorrência

- concorrência
  - modelo de linhas "simultâneas" de execução
  - passam a ser realmente simultâneas na presença de multicores
- concorrência e distribuição
  - aplicações distribuídas envolvem a execução concorrente de processos em várias máquinas
  - uso de concorrência local:
    - atendimento a comunicações concorrentes
    - sobreposição de comunicação e processamento

PUC RIO

alternativas p/ concorrência

1. multithreading
   - várias linhas de execução compartilham globais com escalonamento preemptivo
   - surgido de estudos de sistemas operacionais
   - dificuldades de sincronização
   - exemplos: threads em C (posix) e em Java

exemplo em Java

```
public class ThreadsDorminhocas {
 public static void main(String[] args) {
  new ThreadDorminhoca("1");
  new ThreadDorminhoca("2");
  new ThreadDorminhoca("3");
  new ThreadDorminhoca("4");
 }
}
```

threads em Java

```
class ThreadDorminhoca extends Thread {
 int tempo_de_sono;
 public ThreadDorminhoca(String id) {
  super(id);
  tempo_de_sono = (int) (Math.random() * 5000);
  System.out.println("Tempo de sono da thread "+id+
           ": "+tempo_de_sono+"ms");
  start();
 }
 public void run() {
  try {
   sleep(tempo_de_sono);
  } catch(InterruptedException exception) {
   System.err.println(exception);
  }
  System.out.println("thread "+getName()+" acordou!");
 }
}
```

transferência de controle implícita!
(preempção)

PUC RIO

threads e Java– variáveis compartilhadas

```
class ThreadDorminhoca extends Thread {
 int tempo_de_sono; Conta minhaConta;
 public ThreadDorminhoca(Conta c) {
  minhaConta = c;
  start();
 }
 public void run() {
  // realiza um monte de operações envolvendo minhaConta;
}
```

condições de corrida

```
class Conta {
 private int saldo;
 public Conta (int ini) {
  saldo = ini;
 }
 public int veSaldo() {
  return saldo;
 }
 public void deposita(int dep) {
   saldo = saldo+dep; // problema!!! – difícil de observar
 }
}
```

condições de corrida

```
class Conta {
 private int saldo;
 public Conta (int ini) {
  saldo = ini;
 }
 public int veSaldo() {
  return saldo;
 }
 public void deposita(int dep) {
  for (int i=0; i<dep; i++) { // artificial!! – facilita observação
   saldo++;
  }
 }
}
```

condições de corrida

```
class Conta {
 private int saldo;
 public Conta (int ini) {
  saldo = ini;
 }
 public int veSaldo() {
  return saldo;
 }
public void deposita(int dep) {
  for (int i=0; i<dep; i++) {
   try {
    Thread.sleep(10); //+ artificial – facilita observação
   }
   catch(InterruptedException exception) {
    System.err.println(exception);
   }
   saldo++;
  }
 }
}
```

condições de corrida

```
class Conta {
 private int saldo;
 public Conta (int ini) {
  saldo = ini;
 }
 public int veSaldo() {
  return saldo;
 }
 synchronized public void deposita(int dep) {
  for (int i=0; i<dep; i++) {
   try {
    Thread.sleep(10); //+ artificial:  para escalonador agir!
   }
   catch(InterruptedException exception) {
    System.err.println(exception);
   }
   saldo++;
  }
 }
}
```

- }

alternativas ao modelo multithread clássico

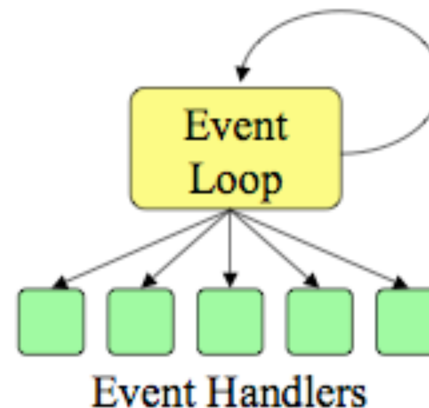
2. modelos orientados a eventos
    - cada evento tratado até o final
    - programa como máquina de estado

alternativas p/ concorrência

- eventos - descrição Ousterhout:

Event-Driven Programming

- **One execution stream: no CPU concurrency.**
- **Register interest in events (callbacks).**
- **Event loop waits for events, invokes handlers.**
- **No preemption of event handlers.**
- **Handlers generally short-lived.**



Why Threads Are A Bad Idea

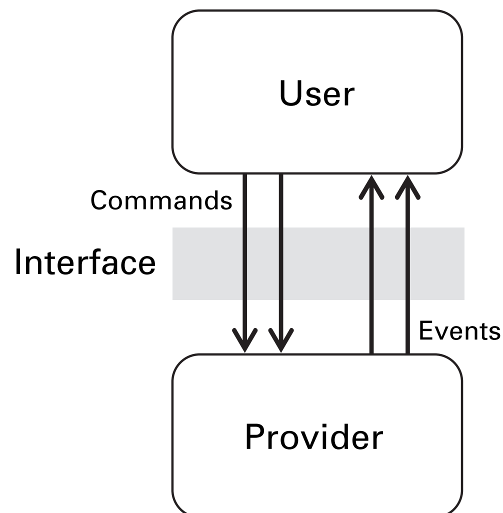
September 28, 1995, slide 9

tratadores de eventos

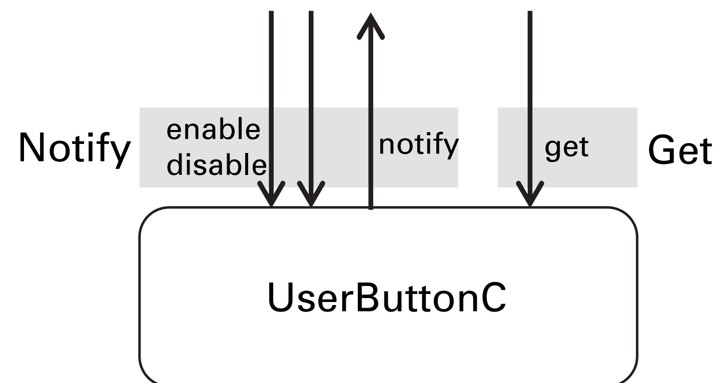
- lógica “invertida”: programa reage a eventos
- conceito geral com várias variantes
  - canal de eventos
  - bindings diretos entre geradores e consumidores

tinyOS

- modelo do sistema é comunicação por comandos e eventos
  - comandos disparam ações que podem retornar através de eventos




```
interface Notify<val_t> {
  command error_t enable();
  command error_t disable();
  event void notify(val_t val);
}
```



eventos no teenyLime

```
1   command result_t StdControl.start() {
2     tuple tempTemplate = newTuple(2, actualField_uint16(TEMPERATURE),
3                                      greaterField(TEMPERATURE_SAFETY_THRESHOLD));
4     call TS.addReaction(TRUE, TL_NEIGHBORHOOD, &tempTemplate);
5     return SUCCESS;
6   }
7    event result_t TS.tupleReady(TLOpId_t operationId,
8                                 tuple *tuples, uint8_t number) {
9      // Notification triggered ...
10  }
```

Fig. 17. TeenyLIME code for an actuator node interested in temperature values.

```
1   command result_t StdControl.start() {
2     return call SensingTimer.start (TIMER_REPEAT, SENSING_TIMER);
3   }
4   event result_t SensingTimer.fired() {
5     return call TemperatureSensor.getData();
6   }
7   event result_t TemperatureSensor.dataReady(uint16_t reading){
8     tuple temperatureValue = newTuple(2, actualField_uint16(TEMPERATURE),
9                                          actualField_uint16(reading));
10    call TupleSpace.out(FALSE,TL_LOCAL,&temperatureValue);
11    return SUCCESS;
12  }
```

Fig. 18. TeenyLIME code for a temperature node.

alternativas ao modelo multithreading clássico

3. multitarefa sem preempção
   - co-rotinas

co-rotinas simétricas: Modula-2

```
MODULE M;
 CONST
  WKSIZE = 512;
 VAR
  wkspA, wkspB : ARRAY [1..WKSIZE]
     OF BYTE;
  main, cA, cB : ADDRESS;
  x : ADDRESS;

 PROCEDURE A;
 BEGIN
  LOOP
   ...
   TRANSFER(x,x);
   ...
  END;
 END A;

 PROCEDURE B;
 BEGIN
  LOOP
   ...
   TRANSFER(x,x);
   ...
  END;
 END B;

 BEGIN (* M *)
   (* create two processes out of
procedure A and B *)
  NEWPROCESS( A, ADR(wkspA),
WKSIZE, cA );
  NEWPROCESS( B, ADR(wkspB),
WKSIZE, cB );
  x := cB;
  TRANSFER(main,cA);
 END M;
```

co-rotinas assimétricas: Lua

```
function boba ()
 for i=1,10 do
  print("co", i)
  coroutine.yield()
 end
end
co = coroutine.create(boba)

coroutine.resume(co) -> co 1
coroutine.resume(co) -> co 2
...
coroutine.resume(co) -> co 10
coroutine.resume(co) -> nada... (acabou)
```

transferência de controle explícita!

menos problemas com condições de corrida!

por outro lado...

- só usamos um processador
- temos que controlar a transferência

PUC RIO

threads + eventos

- sistemas híbridos procuram combinar vantagens de threads e eventos

- tarefa: ler os dois trabalhos abaixo. Escrever um relatório de duas ou três páginas comentando os artigos e, especificamente, como se relacionam com aplicações no ambiente do TinyOS. A data de entrega é 25/3 (por email, em pdf).
  - Douglas C. Schmidt . Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events (1995).
  - Thomas D. Jordan, Irfan Pyarali, Tim Harrison e Douglas C. Schmidt. Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events (1997)

PUC RIO

alternativas ao modelo multithreading clássico

4. multithreading com troca de mensagens
   - Erlang

erlang

- troca de mensagens:
- envio:

```
pid! msg
```

- recebimento:

```
receive
    padrão -> ação
end
```

erlang

```
loop(... ) ->
 receive
  {From, Request} ->
    Response = F(Request),
    From ! {self(), Response},
    loop(...)
 end.
```

Referências

- notas de aula Ihor Kuz, Manuel M. T. Chakravarty & Gernot Heiser (intro-notes.pdf, na página do curso*)
- J. Ousterhout. Why threads are a bad idea (for most purposes)
- von Behren, R., Condit, J., and Brewer, E. 2003. Why events are a bad idea (for high-concurrency servers). In Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9 (Lihue, Hawaii, May 18 - 21, 2003).
- L. Mottola and G.-P. Picco. 2011. Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Comput. Surv. 43, 3, (April 2011) seções 1 e 2
- capítulo de co-rotinas - Roberto Ierusalimschy. Programming in Lua. lua.org, 2013 (1ª edição disponível em www.lua.org/pil/).