Algoritmos e Sistemas Distribuídos

Aula teórica 2

Ano lectivo 2014/2015

Mestrado Integrado em Engenharia Informática

O que é um sistema distribuído?

- Resposta 1: “A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.” -- L. Lamport

Porque usamos sistemas distribuídos?

- Concorrência: N máquinas → N vezes a capacidade computacional
- Tolerância a falhas: Se t<N máquinas deixarem de funcionar, o sistema pode prosseguir

Blessing or curse?

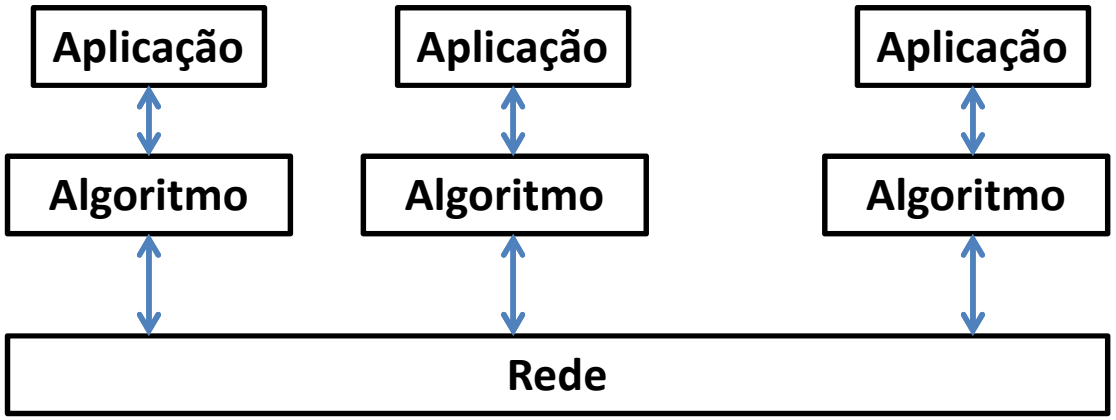
- Desafios na construção de sistemas distribuídos:
- Concorrência: Temos de considerar as possíveis ordens de ocorrência de eventos
- Falhas: Necessário prever a possibilidade um sub-conjunto das máquinas deixar de funcionar

Sistemas concorrentes vs. sistemas distribuídos

- Noutras cadeiras estudarão concorrência
- Área complementar:
  - ambas estudam processos que se executam de forma concorrente
  - técnicas são semelhantes
  - mas, comunicação é através de memória partilhada e não através de mensagens
  - outra diferença fundamental é a necessidade de precaver faltas de um sub-conjunto dos processos

Algoritmos distribuídos

- Servem de base à construção de sistemas distribuídos
- Podem ser encapsulados em abstrações
  - úteis para a construção de outras aplicações/sistemas
  - mais ricas do que as abstrações fornecidas pela camada inferior (rede)



- Como exprimir, analisar, e raciocinar sobre um algoritmo distribuído?

O que é um sistema distribuído? Resposta 2: Modelo

- Modelo de sistema distribuído:
- Processos – elementos computacionais
  - Abstraem noção de máquina/nó
- Rede – Grafo **G=(V,E)**, em que **V** é o conjunto de processos, **E** representa os canais de comunicação entre pares de processos
  - Em geral, vamos considerar um grafo completo, todos os processos estão ligados por arestas bidirecionais

Modelo de sistema distribuído (cont.)

- Mensagens – trocadas entre os processos, pertencentes a um alfabeto **M** + mensagem especial ***null*** representa ausência de mensagem
- Notação: $send_i(j,m)$
  - Processo i envia ao processo j a mensagem m

Modelo temporal (“timing”)

- Define pressupostos sobre questões temporais
- Dois modelos fundamentais:
- Sistema síncrono
  - Existe um limite superior conhecido do tempo de entrega de mensagens pela rede, e respectivo processamento pelos processos
- Sistema assíncrono
  - Não são feitos pressupostos sobre o tempo que demora a entregar mensagens

Modelo de processo

- Identificador único de cada processo: $\mathbf{i} \in \mathbf{V}$
- Cada processo consiste em:
  - $states_i$ – conjunto de estados
  - $start_i$ – estado(s) inicial(is)
  - Inputs e outputs são variáveis especiais no estado
  - $init_i$ – estado inicial
  - Autómato determinístico

Transições entre estados

- Modelo síncrono: execução em rounds
- Funções que determinam transição:
  - $trans_i$: $estados_i$ x vector de M U {null} → $estados_i$
  - $msgs_i$: $estados_i$ → vector de M U {null}
- Em cada round:
  - Aplicar $msgs_i$ para determinar mensagens a enviar
  - Enviar e receber mensagens de todos
  - Aplicar $trans_i$ para determinar estado seguinte

Transições entre estados

- No modelo assíncrono, a diferença é que não há noção explícita de rounds
  - Não se espera por mensagens de todos para fazer a transição
  - Cada mensagem recebida leva a novo estado e um novo conjunto de mensagens enviadas
  - Necessário adaptar definições (fica como exercício)

Modelo de execução (síncrono)

- Execução: descreve a forma como operou uma instância de um algoritmo distribuído
- C – vector de estados (indexado por i $\in$ V)
- M,N – vectores de mensagens ou null, enviadas e recebidas, respectivamente, por cada par de processos
- Execução = C0, M1, N1, C1, M2, N2, C2, …
- Sequência infinita
  - Embora possamos considerar prefixos
- Todos os processos recebem mensagens/ enviam mensagens/transitam de estado em simultâneo
  - Sistema assíncrono: necessário adaptar definição de execução para ter um envio/recepção/transição de cada processo de cada vez

Noção de “trace” (traço)

- Subsequência da execução contendo apenas os inputs e outputs do sistema
  - Relembrar que estes são variáveis do estado
- Modela o comportamento externo do sistema
- Esconde o estado interno
- Notação (sistema assíncrono): para cada elemento da sequência de inputs/outputs, o processo que faz o input ou o output, seguido de “:”, seguido do input ou output correspondente. Exemplo:

  p1:prop(retirar), p2:prop(atacar), p1:decide(atacar), p2:decide(atacar)

Diagrama temporal de uma execução (modelo síncrono)

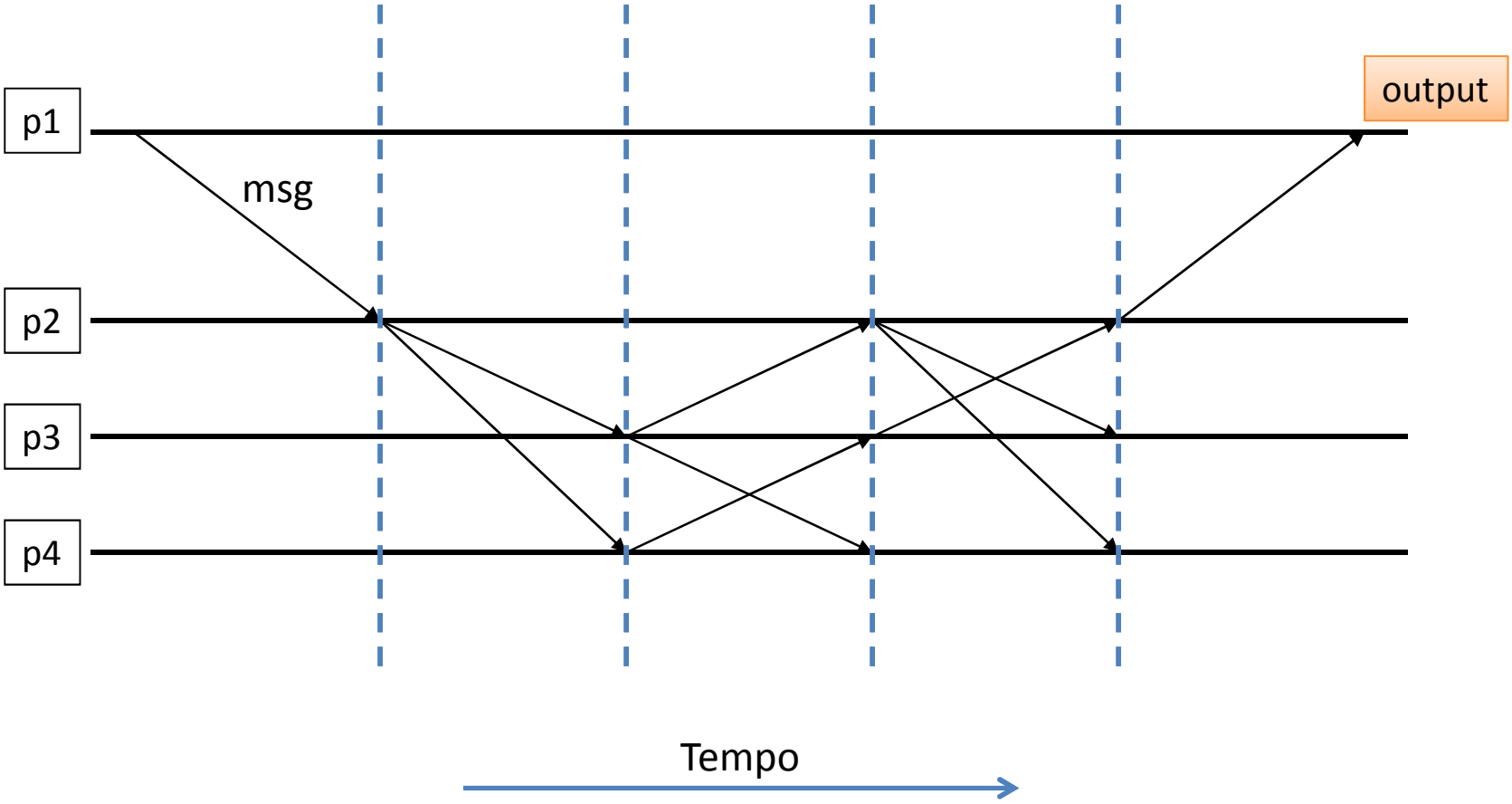
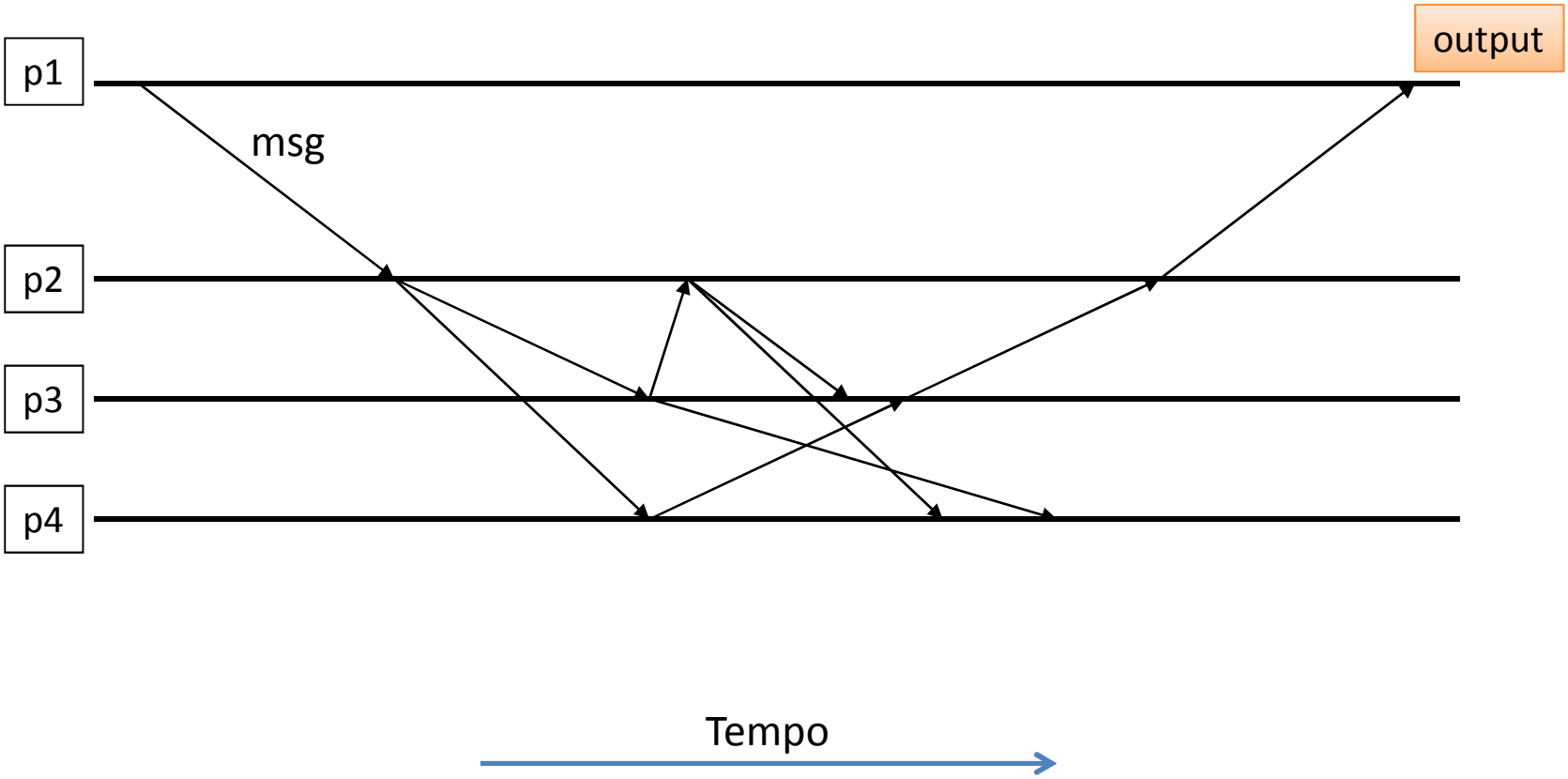


Diagrama temporal de uma execução (modelo assíncrono)



Modelos de falhas (dos processos)

- Processos podem ter falhas que levam a desviar do comportamento esperado
- Terminologia clássica: falta → erros → falhas
  - Vamos usar apenas "**falha**" para referir a um processo que se desvia do comportamento esperado, e dizer que um processo que não falha se denomina "**correcto**"
- Modelos diferentes fazem pressupostos diferentes
- Vamos usar principalmente o modelo de falhas "crash"
  - A partir do momento da falha, processo deixa de enviar mensagens
  - Crash pode ocorrer aquando do envio de msg a vários processos →um sub-conjunto das mensagens é enviado

Outros modelos de falhas

- Modelo de falhas por omissão
  - Processos podem omitir o envio ou a recepção de um sub-conjunto das mensagens
- Modelo “fail-stop”
  - Idêntico ao crash mas ao parar notificam os restantes processos
  - Como simular um modelo fail-stop num sistema síncrono com falhas “crash”?
- Não vamos abordar estes dois modelos

Outros modelos de falhas

- Falhas Bizantinas
  - Comportamento arbitrário dos processos que falham
  - Por exemplo, podem inventar mensagens, alterar valores, duplicar mensagens
  - Modela, por exemplo, bugs de software, corrupção de memória, e também um atacante que controla o processo
  - Dedicaremos uma aula a este modelo
    - Até lá, falha == crash, correcto == não “crasham”

Modelo de rede

- Define pressupostos adicionais sobre os canais de comunicação: possibilidade de perder/duplicar/corromper mensagens
- Modelo justo com perdas (fair-loss):
  - FL1 [fair-loss] Dados processos i,j correctos, se i envia infinitamente m para j, então j recebe infinitamente m
  - FL2 [duplicação finita] Se i envia um número finito de vezes m para j, então j recebe m um número finito de vezes
  - FL3 [ausência de criação] m não é recebida se não tiver sido enviada

Modelo de rede (cont.)

- Modelo teimoso (stubborn)
  - SL1 [entrega teimosa] Se i não falha e envia m para j, então j recebe m infinitas vezes
  - SL2 [ausência de criação] m não é recebida se não tiver sido enviada

- Como implementar modelo stubborn sobre um canal fair-loss?

Modelo de rede (cont.)

- Modelo de canais fiáveis (reliable links)
  - RL1 [validade] Se i,j não falham, qualquer mensagem m enviada de i para j é recebida por j a partir de certo momento (“eventually”)
  - RL2 [ausência de duplicação] Se a mensagem m tiver sido enviada uma única vez é recebida uma única vez
  - RL3 [ausência de criação] m não é recebida se não tiver sido enviada
- Como implementar reliable links sobre um canal stubborn?

Modelo de rede utilizado na cadeira

- Vamos pressupor o modelo de links fiáveis
  - Porque simplifica os algoritmos, ao garantir que as mensagens entre processos correctos não se perdem
  - Para além disso, pode ser implementado sobre os outros
- Notar que estamos ainda sujeitos à escolha síncrono vs. assíncrono
  - Em particular, apesar do nome “fiável”, não temos garantias sobre **quando** é que a mensagem é recebida (no caso do model assíncrono)

Especificações

- Determina as condições que constituem uma solução correcta do problema
- Conjunto de restrições sobre os “traces” que são permitidos
- Dividida em condições de “safety” (segurança) e “liveness” (progresso)

Safety

- Condições que se devem verificar em qualquer instante da execução
  - Ou seja, outputs “maus” que têm de ser sempre evitados
- Características:
  - Um trace vazio obedece às condições de safety
  - Um prefixo de um trace que obedece a safety também obedece a safety

Liveness

- Condições que se devem verificar a partir de um certo ponto na execução
  - Ou seja, outputs “bons” que deverão a certa altura ocorrer (em inglês, “eventually”)
- Características:
  - É sempre possível criar uma extensão de um trace de forma a este obedecer a uma condição de liveness

- Nalguns casos as condições podem ter aspectos de safety e de liveness
  - Separáveis na conjunção de várias sub-condições

Exercício

- Para o problema dos dois generais da aula anterior, idenficar condições de safety e de liveness
- Alterar uma condição de liveness de forma ao problema ter solução (assumir canal justo com perdas)
- Alterar uma condição de safety de forma ao problema ter solução
- Esboçar a solução dos pontos anteriores

Revisão do problema dos dois generais (Retirados pressupostos sobre modelo)

- Ambos os generais têm uma preferência inicial (atacar ou retirada) e não podem discordar na (possível) decisão final
- Se ambos tiverem a mesma preferência essa é a única decisão possível
- Os generais que não são fulminados têm sempre de decidir (em tempo finito)

Soluções

- [S] Ambos os generais têm uma preferência inicial (atacar ou retirada) e ~~não~~ podem discordar na (possível) decisão final
- [S] Se ambos tiverem a mesma preferência essa é a única decisão possível
- [L] Os generais que não são fulminados têm ~~sempre~~ de decidir (em tempo finito) sempre que não hajam crashes