Algoritmos e Sistemas Distribuídos

Aula teórica 3

Ano lectivo 2014/2015

Mestrado Integrado em Engenharia Informática

Na última aula

- Modelo de sistema distribuído
- Modelo síncrono vs. assíncrono
  - Qual a diferença?
- Apesar da assincronia, vamos considerar o modelo de canais fiáveis (reliable links)
  - Mensagens enviadas entre processos correctos são entregues a partir de certo momento

Diagrama temporal de uma execução (modelo síncrono)

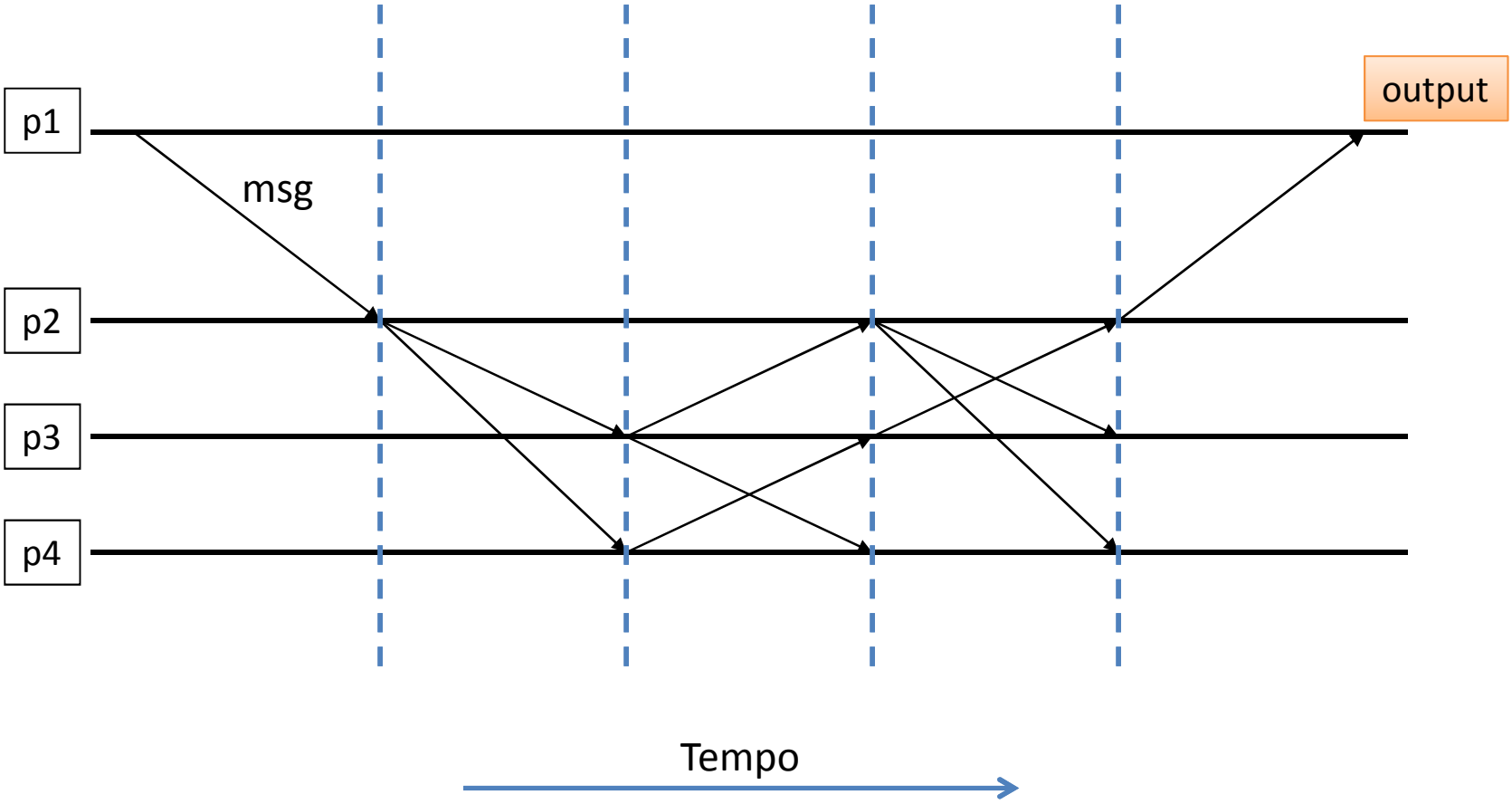


Diagrama temporal de uma execução (modelo assíncrono)

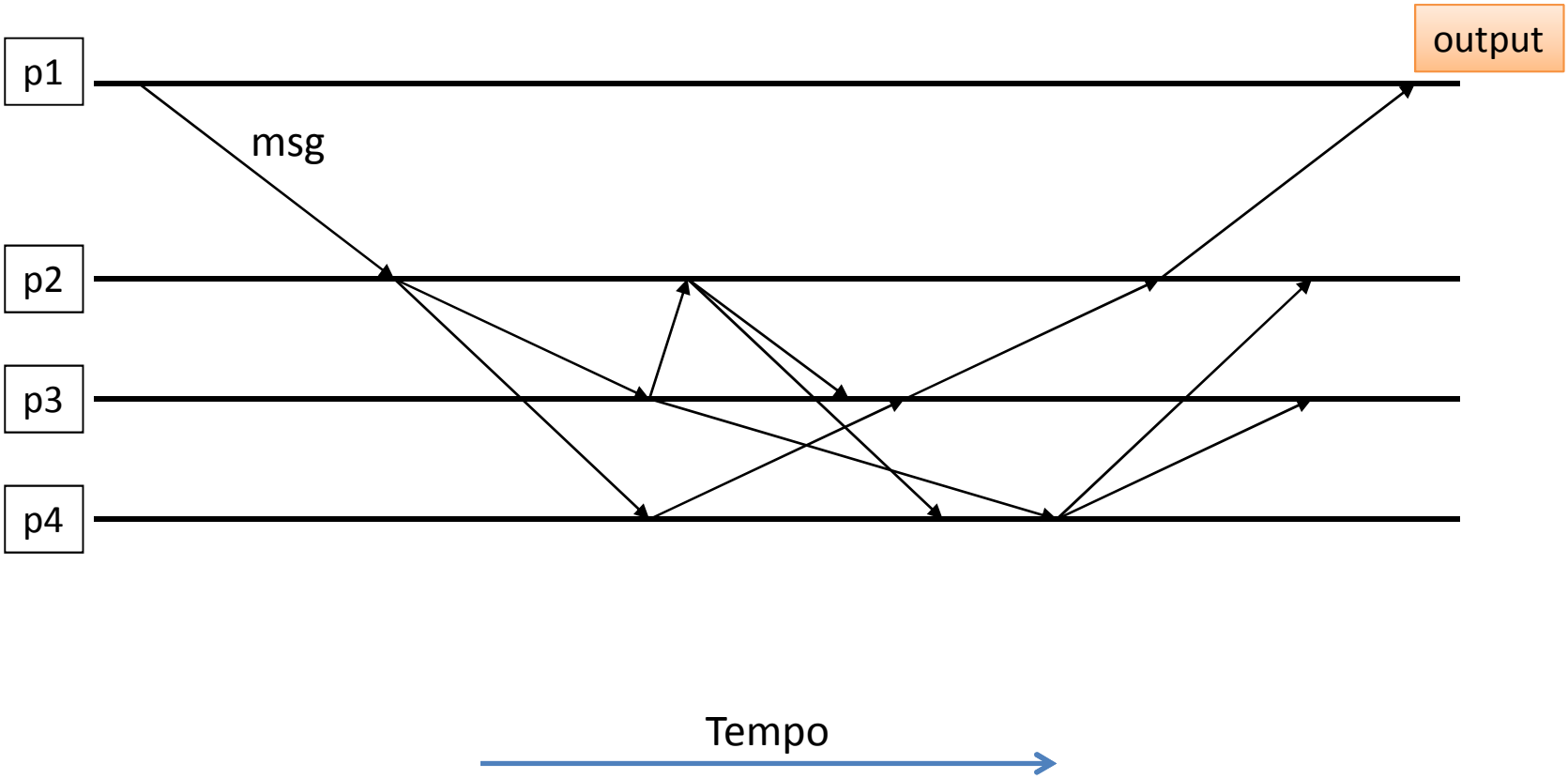
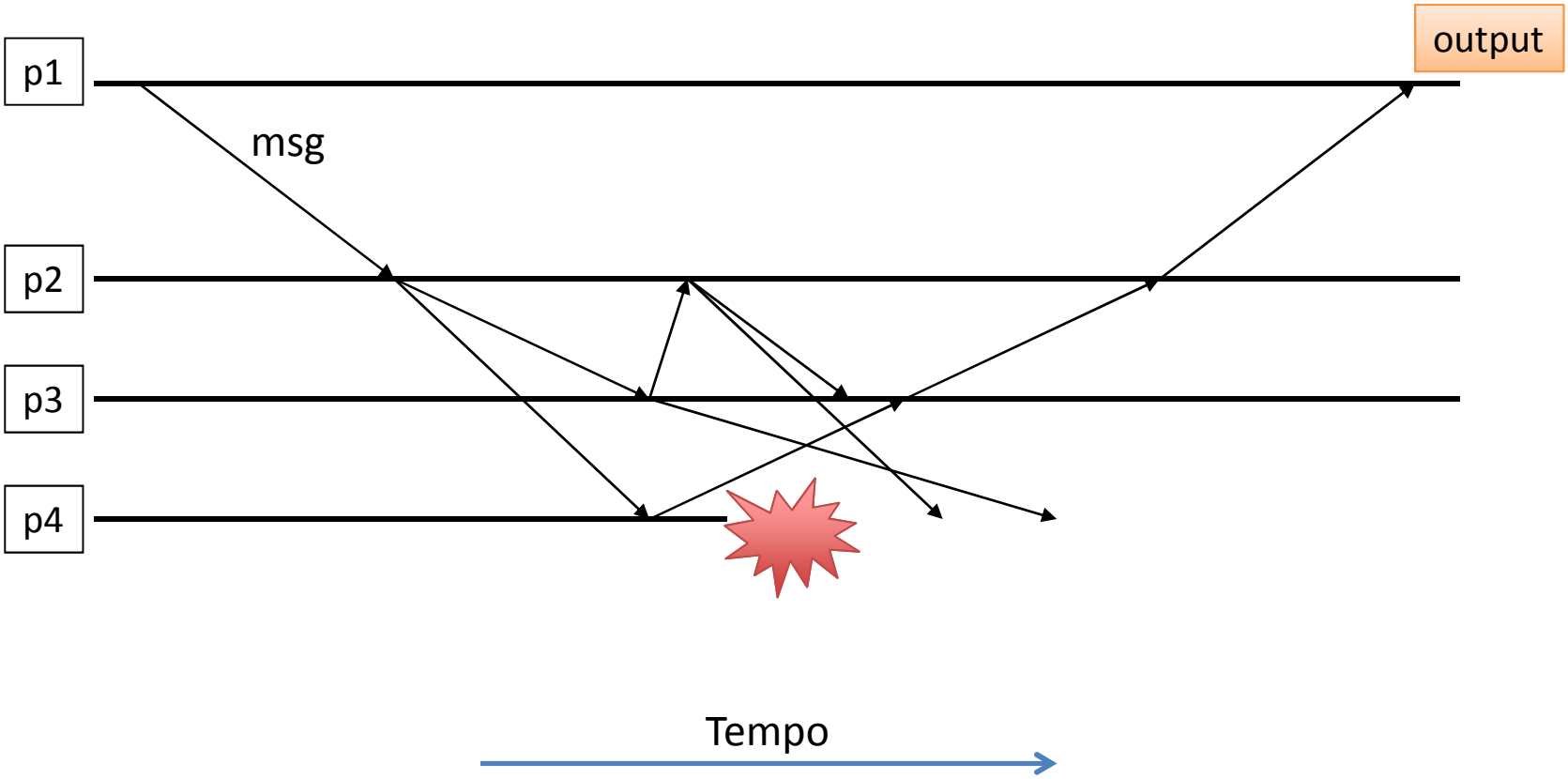


Diagrama temporal de uma execução (modelo assíncrono com “crashes”)



Safety vs. Liveness

- Safety: Condições que se devem verificar em qualquer instante da execução
  - Ou seja, outputs “maus” que têm de ser sempre evitados
- 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”)

Métodos para provas: invariantes

- Método importante para demonstrar a correcção de um algoritmo
- Invariante – condição sobre o estado do sistema que é verdadeira em qualquer estado atingível
  - Estado atingível – aparece em alguma execução do sistema, ou seja, é possível chegar a esse estado a partir do estado inicial
- Normalmente prova-se um invariante por indução no número de passos de uma execução

Problema da difusão (broadcast)

- Processo que precisa de comunicar a mesma mensagem a N outros processos
- Garantir entrega coerente entre os processos que não falham (todos entregam ou nenhum entrega)
- Há várias variantes, vamos considerar apenas a difusão fiável (reliable broadcast)

Especificação do problema

- Inputs: um processo é designado emissor e tem como input na variável init-val_i a mensagem *m* a transmitir, os restantes têm a variável de input init-val_i a *null*
- Outputs: cada processo tem como saída uma variável $decision_i$, inicialmente *null*
- RB1 [Validade] Se o emissor não falhar e tiver como input *m*, todos os processos correctos fazem a certo ponto output de *m*
- RB2 [Consenso] Se um processo correcto fizer output de *m*, todos os processos correctos fazem a certo ponto output de *m*
- RB3 [Integridade] Todos os processos decidem num único valor, e, se este for *m*, então o input do emissor foi *m*

Exercício:
encontrar solução para o problema

- Pressupor modelo assíncrono, falhas por crash, canais fiáveis

Solução #1 (errada)

$states_i$:
  $init\text{-}val_i$ // input, mensagem a transmitir ou null
  $decision_i$ // output, mensagem entregue, inicialmente null

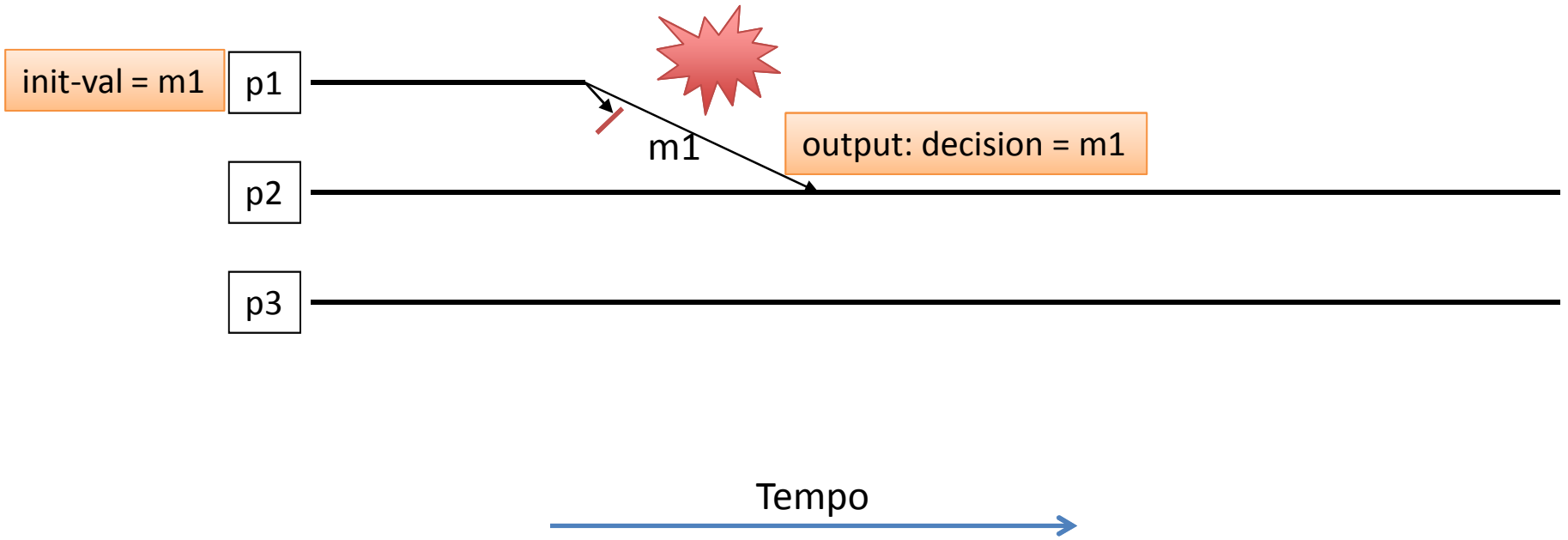
$trans_i$: definido pelo pseudo-código:

```
initialization:
    if (init-val_i != null) {
        forall (j in V)
            send_i(j, init-val_i);    // enviar a todos
    }

upon receive_i(m) from j:
    if (decision_i == null)
        decision_i = m;    // output m
```

Não basta o emissor fazer o envio da mensagem para todos, porquê?

Qual a propriedade que não foi obedecida nesta execução?



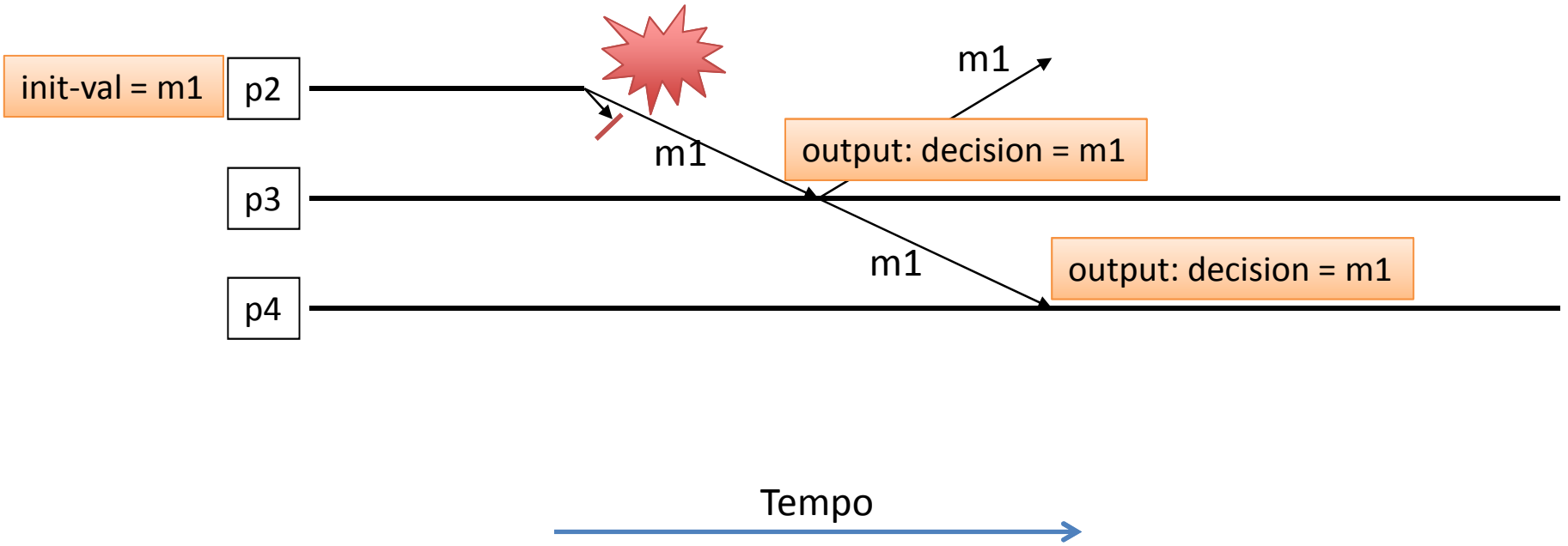
Solução #2 (correcta)

initialization:

```
    if (init-val_i != null && decision_i==null) {
            forall (j in V)
                send_i(j, init-val_i);    // enviar a todos

upon receive_i(m) from j:
    if (decision_i == null) {
            decision_i = m;    // output m
            forall (k in V-{j});
                send_i(k, m);    // ecoar (retransmitir) para todos
    }
```

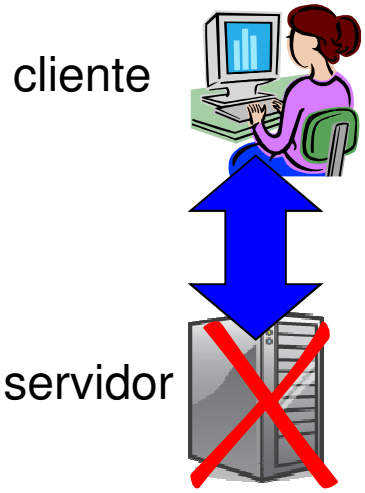
Execução da solução #2

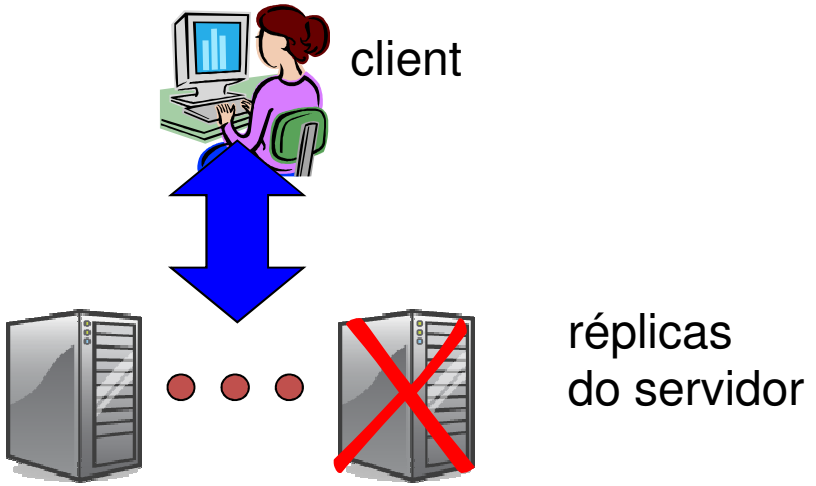


Correcção (argumento informal)

- RB1 [Validade] Se o emissor não falhar e tiver como input *m*, envia *m* a todos, e pela propriedade dos canais fiáveis e pelo pseudo-código todos entregam *m*
- RB2 [Consenso] Se um processo correcto entregar *m* então vai enviar a todos, e como não falha e pela propriedade dos canais fiáveis todos os correctos entregam *m*
- RB3 [Integridade] Se input é *m* então *m* é o único valor possível nas mensagens, logo o único output possível

Problema: replicar um servidor





Servidor == Máquina de estados

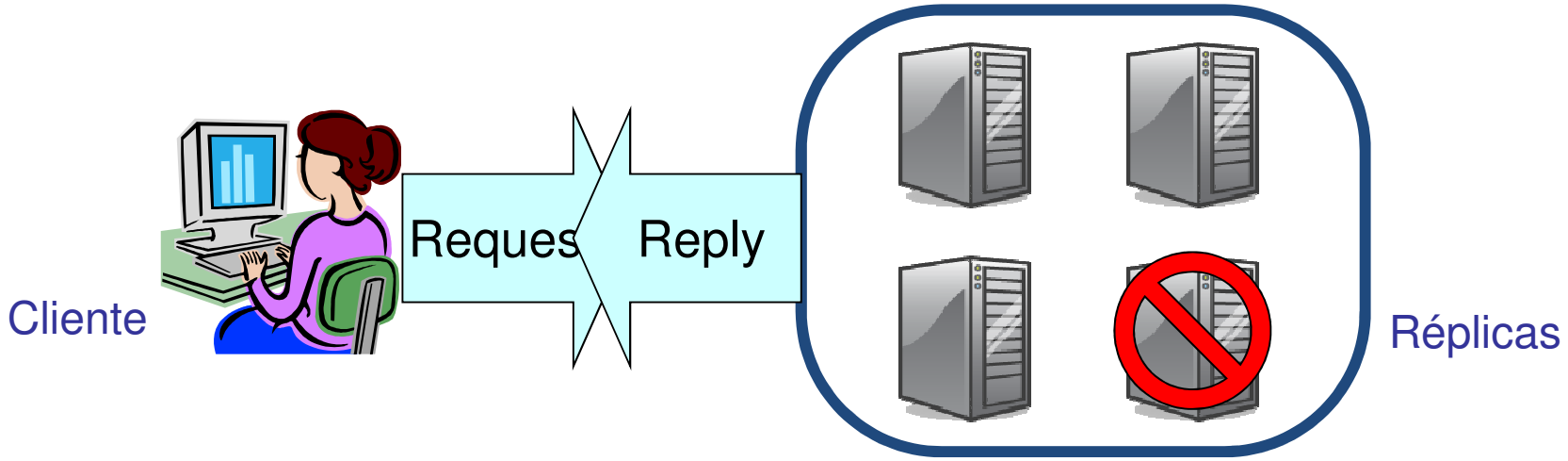
- Cada servidor corre uma cópia do serviço que:
- Mantem um estado **S** (memória, conjunto de variáveis)
- Exporta um conjunto de operações **O**
- Cada operação tem argumentos (inputs), e:
  - Gera uma resposta (output)
  - Servidor transita para estado seguinte

Não-determinismo

- Uma operação (do serviço que corre em cada servidor) é não-determinística se o estado ou a resposta que produz não depende exclusivamente do conteúdo do pedido e do estado inicial
- Não-determinismo complica substancialmente a replicação de um servidor
  - Estado das réplicas vai divergir, levando a respostas contraditórias

Replicação de máquinas de estados (State machine replication = SMR)

1. Tornar o serviço determinístico (máquina de estados)
2. Replicar o servidor
3. Garantir que as réplicas correctas seguem a mesma sequência de transições entre estados
4. Votar nos outputs de forma a tolerar falhas



Requisito central da SMR

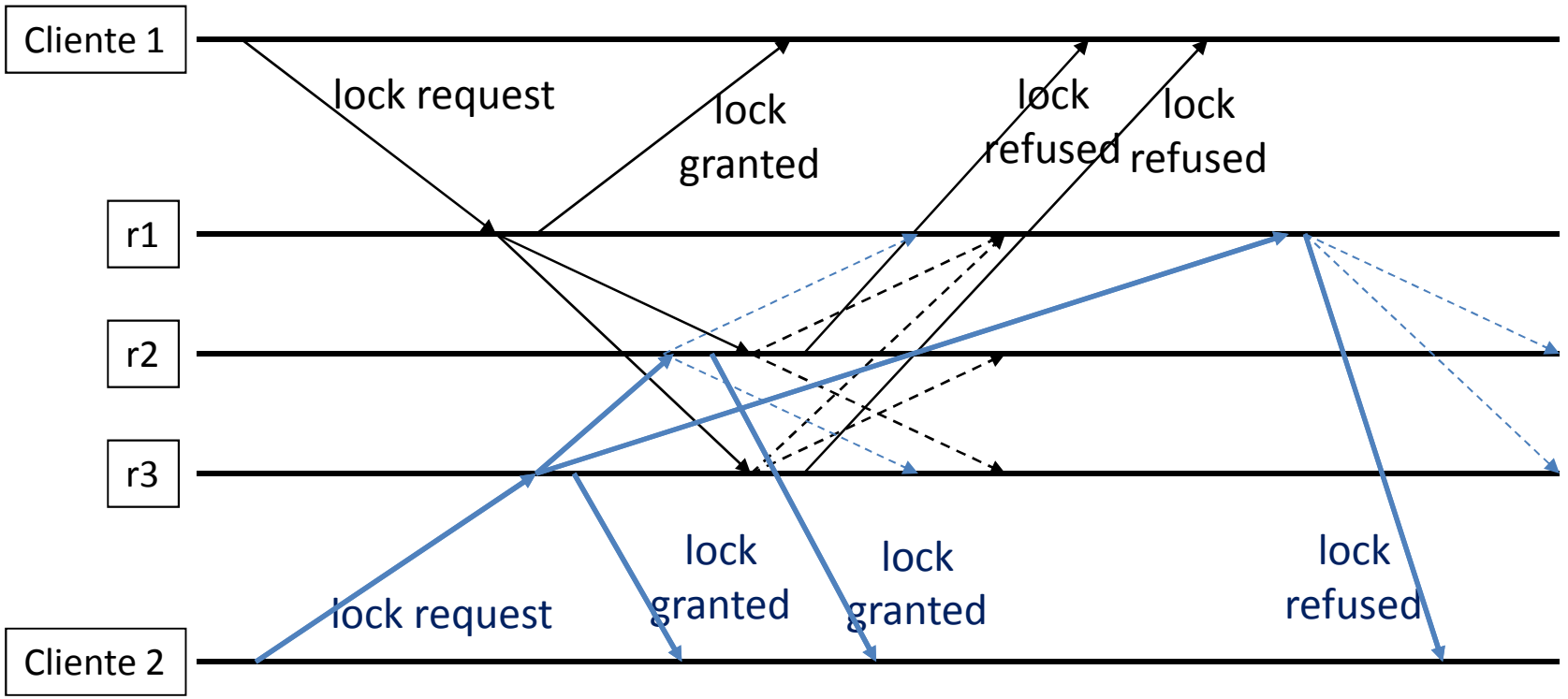
Todas as réplicas correctas têm de receber todas as operações pela mesma ordem

- Tolerância a falhas:
  - Número de réplicas é parametrizado pelo número de falhas que são toleradas
  - Por exemplo, n=2f+1 réplicas para tolerar f falhas
  - Valor de n depende do algoritmo usado, dos pressupostos do modelo, etc.

Limitações da difusão fiável

- Como utilizar a difusão fiável para implementar replicação de máquinas de estados?
- Tentativa (falhada):
  - Cada cliente contacta uma réplica (à escolha) que faz o “reliable broadcast” da operação para todas as réplicas
  - Réplicas executam e devolvem resultado ao cliente
  - Cliente espera por um número “suficiente” de respostas
- Porque não funciona?

Replicas não concordam com ordem de execução



É necessária uma abstracção mais poderosa!

- A difusão fiável por si não garante o requisito central: réplicas correctas devem receber as operações pela mesma ordem
- Qual é a abstracção que põe as réplicas de acordo nesta ordem?
  - Voltamos a chamar os generais!

Consenso

- Inputs: cada processo tem como input na variável v_i uma proposta inicial
- Outputs: cada processo tem como saída uma variável $decision_i$, inicialmente *null*
- C1 [Validade] Se todos os processos tiverem $v_i = v$, então v é o único valor do output permitido
- C2 [Acordo] Dois processos correctos não podem decidir valores diferentes
- C3 [Terminação] Todos os processos correctos têm de decidir a certo ponto ("eventually")
- Opcional: C4[integridade] Se um processo correcto decide v, então v era a proposta inicial de algum processo

Nota: definição tem de ser adaptada para o caso de falhas Bizantinas

Perguntas, perguntas, perguntas

- O problema é solúvel? Sim? Não? Depende das circunstâncias?
- Se depende, então quais os pressupostos necessários? Existe solução quer no modelo síncrono quer no modelo assíncrono?
- Qual o número de processos necessários para resolver este problema?
- Qual a complexidade da solução em termos de tempo e número de mensagens?

Próximas aulas…

- Solução para o consenso num sistema síncrono
- Resultados de impossibilidade:
  - Número mínimo de rounds para resolver o consenso num sistema síncrono
  - Impossibilidade de resolver o consenso em sistemas assíncronos
- Solução para o consenso num sistema (praticamente) assíncrono: Paxos

Modelo

- Síncrono
- Links fiáveis
- $n >= f + 1$ processos
- Falhas por “crash” de até f processos

Solução #1 (errada)

$states_i$:
  v_i - input, proposta inicial
  $decision_i$ - output, decisão final
  $vals_i$ - inicialmente $\{v_i\}$
  $rounds_i$ – inteiro que representa o número do round, inicialmente 0

$msgs_i$:
  if ($rounds_i$==0)
    forall (j in V)
      $send_i(j, \{v_i\})$; // send to all
    }

$trans_i$:
    $rounds_i$=$rounds_i$+1
    forall (j in V-{i})
      receive(v_j) from j
       $vals_i$ = $vals_i$ U $\{v_j\}$
    $decision_i$ = min($vals_i$)

Qual o problema da solução anterior?

Solução #2

$states_i$:
  v_i - input, proposta inicial
  $decision_i$ - output, decisão final
  $vals_i$ - inicialmente $\{v_i\}$
  $rounds_i$ – inteiro que representa o número do round, inicialmente 0

$msgs_i$:
  if ($rounds_i$<=f)
    forall (j in V)
      $send_i$(j, $vals_i$); // send to all
    }

$trans_i$:
    $rounds_i$=$rounds_i$+1
    forall (j in V-{i})
      receive(S) from j
      $vals_i$ = $vals_i$ U S
    if ($rounds_i$ == f+1)
      $decision_i$ = min($vals_i$)

Correcção (ideias chave)

- Lema 1: Se num dado round r não há falhas, então, para quaisquer dois processos i,j que não falharam no final de r, $vals_i = vals_j$
- Lema 2: Se todos i,j têm valsi=valsj ao final de r, então para qualquer r'>r, $vals_i = vals_j$
- Lema 3: Se i,j estão activos ao final de f+1 rounds, nessa altura $vals_i = vals_j$
- Como derivar C1,C2,C3,C4?

Complexidade

- Temporal: f+1 rounds
- Número de mensagens: $O((f+1)n^2)$

Exercício

- Modificar o algoritmo de forma a reduzir o número de rounds em situações em que os processos não falham

- Modificar o algoritmo de forma a evitar que o tamanho de cada mensagem seja O(n)