Consenso

Consenso

- O consenso, o acordo entre os nós de um sistema acerca de um dado valor, é um dos problemas mas difíceis e mais estudado nos sistemas distribuídos.
- Não tem solução no caso geral, nomeadamente num **sistema assíncrono onde podem ocorrer falhas**
- Este resultado de impossibilidade é conhecido pelas iniciais dos autores da descoberta: impossibilidade FLP

Consenso

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

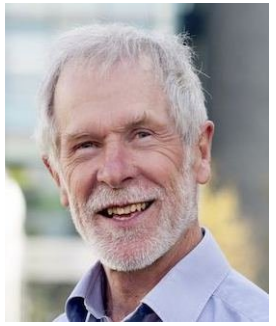
Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.



Consenso

- Conjunto de N processos
- Cada processo propõe um valor (*input*)
- Todos os processos decidem o mesmo valor (*output*)
- O valor decidido deve ser um dos valores propostos:
  - Isto impede uma solução em que se decide sempre um valor por omissão independentemente dos valores propostos.
- Pode ser qualquer um dos valores propostos:
  - Não tem de ser o valor proposto por mais processos.
  - Não há valores melhores que outros.
  - Nem proponentes melhores que outros

Propriedades do consenso

- Terminação: todos os processos correctos decidem (alguma-vez)
- Acordo (uniforme): se dois processos decidem, decidem o mesmo valor
- Integridade: o valor decidido foi proposto por um processo

Soluções para o consenso

- Soluções para sistemas síncronos:
  - Onde é possível concretizar um detector de falhas perfeito

- Soluções para sistemas parcialmente assíncronos
  - Onde é possível concretizar um detector de falhas “alguma-vez” perfeito.
  - Um algoritmo relevante neste contexto é um algoritmo concebido por Lamport e denominado “Paxos”.
  - *Fora da matéria de SD*

Algoritmo para sistemas síncronos

- Existem vários algoritmos para concretizar o consenso em sistemas síncronos
- Vamos apresentar um algoritmo que se designa por “FloodSet”

FloodSet consensus

f é o número máximo de processos que podem falhar

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \leq r \leq f + 1)$

Cada ronda dura o tempo máximo entregar uma mensagem

Difusão básica (não fiável)

B-multicast$(g, Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round *r*)

{

On B-deliver(V_j) *from some* p_j

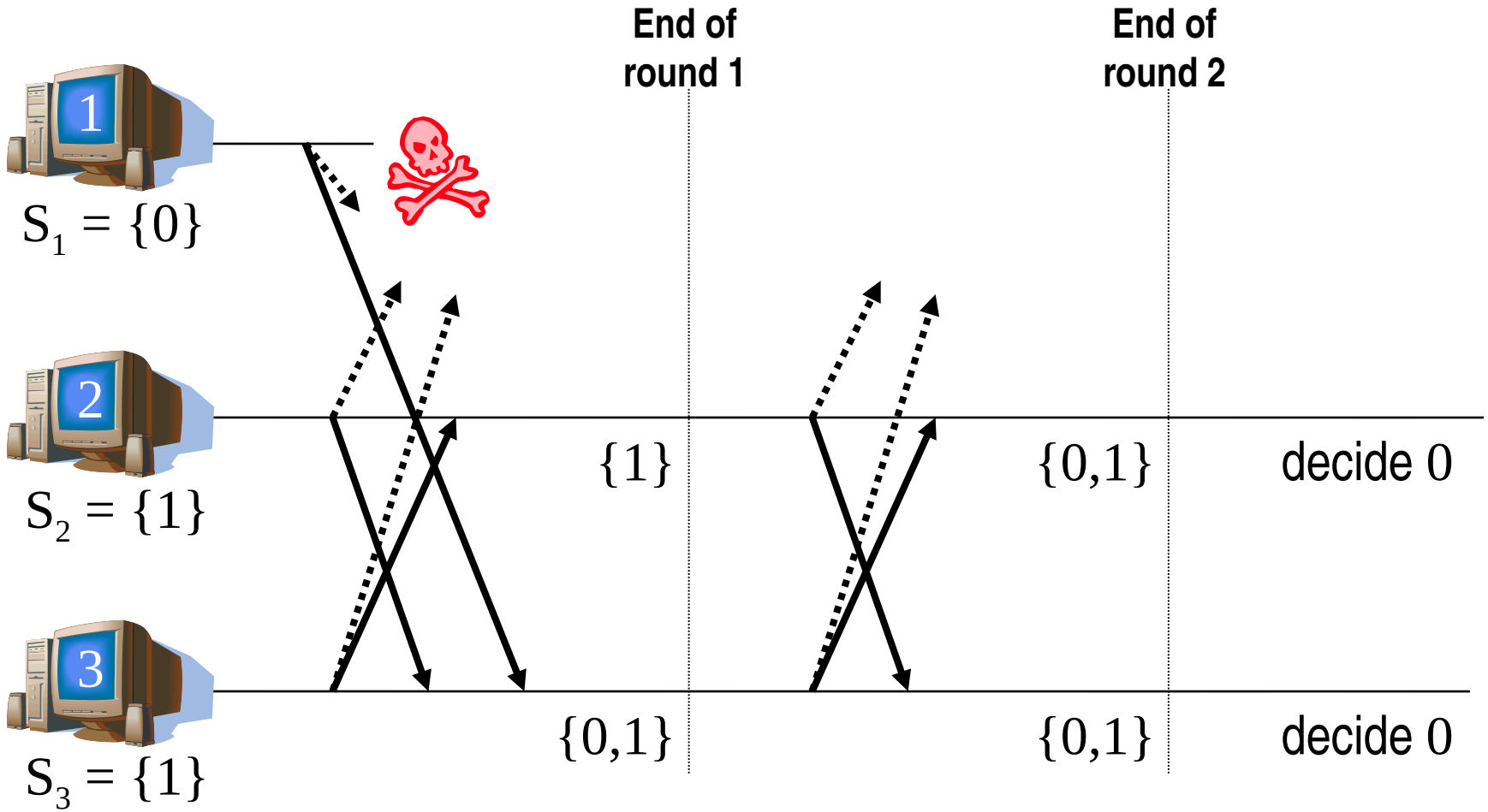
$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f + 1)$ *rounds*

Assign $d_i = minimum(Values_i^{f+1})$;

Execução com *f=1*



FloodSet consensus

- O algoritmo pressupõe que o sistema é sincrono:
  - Se um processo p_i não recebe o valor de outro processo p_j no turno n, então o processo p_j falhou de certeza (e não participa nos turnos posteriores)
- É possível adaptar o algoritmo para usar um detector de falhas perfeito:
  - Um processo p_i não avança para o turno n+1 sem receber o valor do processo p_j no turno n, a não ser que o detector de falhas perfeito declare p_j como falhado
- Nalguns casos, quando não ocorrem falhas, é possível terminar em menos turnos

Consenso e Replicação

Problemas relacionados com consenso

- Ordem total e máquina de estados replicada
- Replicação primário secundário
- Sincronia na vista

Difusão atómica usando consenso

- É necessário realizar uma sequência de instâncias de consenso.
- Cada instância decide um conjunto de mensagens a entregar.

Difusão atómica usando consenso

```
Init:
  por_ordenar = {}
  ordenadas = {}
  num_seq = 0;

Quando AB.send(m):
  RB.send(m)

Quando RB.deliver(m):
  por_ordenar = por_ordenar ∪ {m}

while TRUE
begin
  espera até que (por_ordenar \ ordenadas <> {});
  num_seq = num_seq + 1

  // executa mais uma instância de consenso
  Consensus[num_seq].propose[(por_ordenar \ ordenadas)
  espera ate Consensus[num_seq].decide(proximas)
  // o consenso decide quais a mensagens a entregar

  ordenadas = ordenadas ∪ proximas

  para todas as mensagens m em proximas //por uma ordem determinista
  AB.deliver(m)
end
```

Máquina de Estados Replicada

- Os clientes usam difusão atómica para enviarem os pedidos para todas as réplicas:
- As réplicas recebem todas:
  - O mesmo conjunto de pedido
  - Pela mesma ordem
- A réplicas processam os pedidos pela ordem total definida pela difusão atómica:
  - Computações deterministas
  - Um único fio de execução
- As réplicas respondem aos clientes.

Primário-secundário

- É possível concretizar um sistema de primário-secundário que seja robusta na ausência de um detector de falhas perfeito?

Primário-secundário com consenso

```
pedidos_pendentes = {}
pedidos_executados = {}
propostas = {}
num_seq = 0;
decidido[] = false //para todos os numeros de sequencia


Quando recebe pedido do cliente P
  pedidos_pendentes = pedidos_pendentes ∪ {P}


Quando sou o lider AND
  decidido[num_seq] = true AND
  (pedidos_pendentes \ pedidos_executados <> {})
  begin
    proximo_pedido = escolhe_um (pedidos_pendentes \ pedidos_executados)
    <proximo_estado, resposta_cliente> = ExecutaPedido(estado, proximo_pedido);
    RB.send(<PROPOSTA, my_id, num_seq+1, proximo_pedido, proximo_estado, resposta_cliente>)
  end


Quando RB.deliver(proposta=<PROPOSTA, my_id, num_seq+1, proximo_pedido, proximo_estado, resposta_cliente>)
  propostas = propostas ∪ {proposta}


Quando existe proposta = <PROPOSTA, id, n, pedido, estado> em propostas tal que: decidido[num_seq] = true AND ns = num_seq+1 AND
id = lider
  consenso[num_seq+1].propose(proposta_input = <PROPOSTA, id, ns, pedido, estado, resposta>)
  espera até que consenso[num_seq+1].decide(proposta=<PROPOSTA, id_out, sn_out, pedido_out, estado_out,resposta_out>)

  num_seq = num_seq+1;
  decidido[num_seq] = true;
  estado = estado_out;
  pedidos_executados = pedidos_executados ∪ {pedido_out}
  envia resposta_out para o cliente
```

Sincronia na Vista com Consenso

- A mudança da vista V_i para a vista V_{i+1} consiste em:
  - Recolher todas as mensagens entregues na vista V_i
  - Propôr para consenso um tuplo

    V_{i+1} = <membros de V_{i+1}, conjunto de mensagem entregues em V_i>

  - Esperar o resultado do consenso
  - Entregar as mensagens em falta
  - Entregar a nova vista

Sincronia na Vista com Consenso

- Cada processo p_i mantém um registo h_i de todas as mensagens que já entregou numa dada vista.
- Quando se inicia a mudança de vista de V_i para V_{i+1}, uma mensagem especial “view-change (*saídas, entradas*)” é enviada para todos os processos da vista V_i (a lista de saídas ou de entradas pode ser vazia)
- Quando um processo recebe a “view-change”, pára de entregar novas mensagens, inicia “V_{i+1} = V_i-*saídas+entradas*”, inicia h_j=*null* para todos os processo p_j na V_i e envia h_i para todos os outros processos na V_{i+1}
- Cada processo recebe os valores h_j de todos os outros processos p_j na V_i.

Sincronia na Vista com Consenso

- Se um processo p_i não recebe h_j de um processo p_j e suspeita que p_j falhou, retira p_j da V_{i+1} e coloca $h_j={}$
- Quando p_i tem todos os valores de h_j <> *null (*isto é, se já recebeu h_j de p_j ou se assume que p_j falhou*)* define o conjunto de mensagem a entregar $M\text{-}V_i$ como a união de todos os h_j recebidos e inicia o consenso propondo $<V_{i+1}, M\text{-}V_i>$
- O resultado do consenso define a nova vista e o conjunto de mensagens a entregar na vista anterior