Sistemas Distribuídos
Modelos de Enlaces

Prof. Elias P. Duarte Jr.
Universidade Federal do Paraná (UFPR)
Departamento de Informática
www.inf.ufpr.br/elias/sisdis

Sumário da Aula de Hoje

- Vamos começar relembrando o paradigma de troca de mensagens & primitivas de comunic.
- 2 propriedades importantes: *safety* & *liveness*
- A Mensagem: identificador único
- Modelos de enlaces:
  - *Canal de comunicação fair-loss*
  - *Garantindo a entrega sem duplicação*
  - *Canal de comunicação perfeito*

O Sistema Distribuído

- Um sistema distribuído é um conjunto de *N* processos

  $S = \{p_1, p_2, \ldots, p_N\}$

- Algoritmo distribuído: o mesmo algoritmo é executado por todos os processos localmente
- Processos se comunicam trocando mensagens

2 Tipos de Propriedades

- Quando vamos demonstrar que um algoritmo distribuído “funciona de verdade”
- Provamos sua correção através de propriedades
- As propriedades podem ser classificadas em um de dois tipos:
  - ***Safety***
  - ***Liveness***

Safety

- As propriedades de safety garantem que “nada de ruim acontece”
- Por exemplo: o consenso é o acordo, sobre um único valor
- Uma propriedade de safety de um algoritmo de consenso garante que jamais 2 processos vão decidir por 2 valores diferentes
- Pode ser traduzido como “segurança”, melhor usar *safety* mesmo ;-)

Liveness

- Tem um jeito fácil de garantir a *safety*

Liveness

- Tem um jeito fácil de garantir a *safety*
- Os processos ficam parados e não fazem nada :-)
- Assim eu garanto que jamais 2 processos distintos decidem por 2 valores diferentes
- Claramente insatisfatório :-/
- As propriedades de *liveness* garantem que “algo de bom acontece”
- No caso do consenso: os processos efetivamente decidem!
- Pode ser traduzido como “progressão”

Fairness

- Um terceiro tipo de propriedade é importante em alguns casos
- “Justiça”: todos os processos tem “direitos” iguais
  - Um bom exemplo: exclusão mútua distribuída
  - Garante que em um instante de tempo apenas 1 único processo acessa 1 recurso
  - *Safety* (jamais 2 processos acessam o mesmo recurso simultaneamente); *Progress* (efetivamente algum processo acessa o recurso)
  - *Fairness:* os diferentes processos tem a mesma chance de acessar o recurso (poderia garantir a *safety* e *liveness* estabelecendo que um único processo acessa o recurso)

A Mensagem

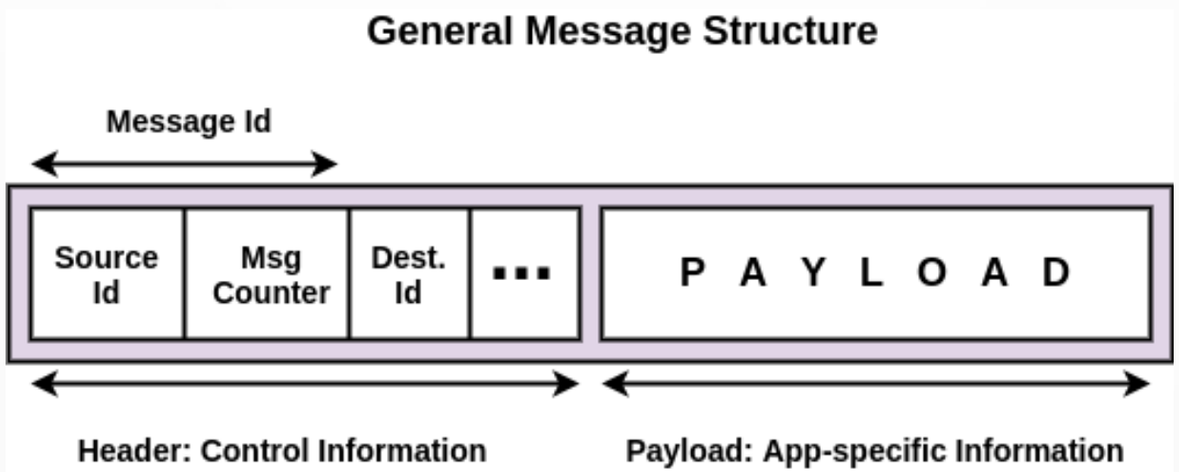
- Processos se comunicam trocando mensagens
- A mensagem tem informações “de controle”: incluindo entre outras possíveis o identificador do processo origem (*source*) e o identificador do processo destino (*destination*)
- Além disso a mensagem carrega informações específicas da aplicação distribuída: antigamente “dados”, hoje “*payload*”

Identificando uma Mensagem

- É surpreendentemente fácil identificar uma mensagem de forma única
- O identificador da mensagem consiste do endereço da origem e contador de mensagens local
- O contador de mensagens é zerado antes da execução do algoritmo; a cada mensagem transmitida: incrementado de 1
- Assim, podemos assumir que cada mensagem carrega seu identificador (i.e., id-origem + contador)

A Mensagem

- Portanto, quando falarmos em “mensagem” pense em uma estrutura assim:



Primitivas de Comunicação

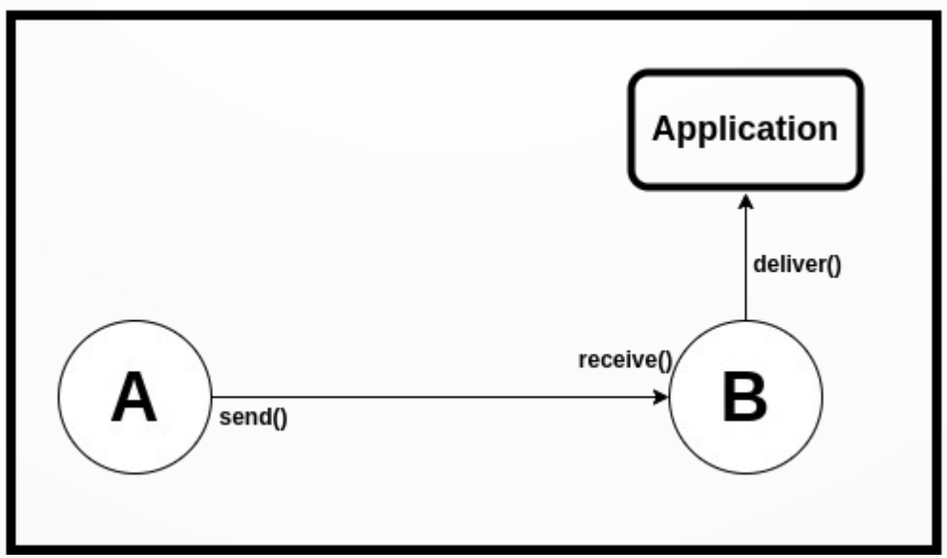
- Nossas primitivas básicas de comunicação são: *send(msg), receive(msg) e deliver(msg)*
- Veja que às vezes vamos explicitar parâmetros para deixar mais claro o que está acontecendo
  - Por exemplo, podemos explicitar na primitiva os endereços de origem e destino:

    send(source-id, dest-id, msg)

    - *Os endereços já estão na mensagem, mas em certos casos fica mais claro explicitá-los na primitiva*

Relembrando o *deliver(msg)*

- Uma mensagem recebida pode ser descartada (por diversos motivos) ou entregue



As Mensagens São Transmitidas através de Canais de Comunicação

- Termos equivalentes usados: *link, link de comunicação, enlace*
- Qual a tecnologia de um canal de comunicação?

As Mensagens São Transmitidas através de Canais de Comunicação

- Qual a tecnologia de um canal de comunicação?
- Qualquer uma! Não importa
  - Wireless, fibra ótica, cabo coaxial...
  - Internet, rede local, ou pode ser mesmo que os processos estejam em execução na mesma máquina
- Basta que os dois processos consigam fazer uma transmissão de mensagem no canal de comunicação
  - A origem usa *send(msg)* para transmitir a mensagem que é recebida pelo destino com *receive(msg)*
  - Uma pilha de protocolos de rede está envolvida na transmissão

Modelos de Canais de Comunicação

- Dois modelos de canal de comunicação são muito importantes:
  - Canais de comunicação *fair-loss*
  - Canais de comunicação perfeitos
    - Podem ser chamados também de confiáveis

Modelos de Canais de Comunicação

- Dois modelos de canal de comunicação são muito importantes:
  - Canais de comunicação *fair-loss*
  - Canais de comunicação perfeitos
    - Podem ser chamados também de *confiáveis*
- Por que são importantes? **Toda** vez que se descreve um sistema distribuído ou um algoritmo distribuído deve ser explicitado o modelo de enlace

Enlaces *Fair-Loss*

- Os enlaces *fair-loss* podem perder mensagens
- Mas existe uma *boa* probabilidade de que uma mensagem transmitida chegue ao destino
  - Reflete enlaces reais IP, por exemplo
  - Pense bem: faz sentido que estes enlaces *fair-loss* tenham uma boa probabilidade de sucesso!

Propriedades dos *Enlaces Fair-Loss*

- Considere que um processo *p* transmite uma mensagem para um processo *q* sobre um canal de comunicação *fair-loss*
- As propriedades desse tipo de enlace são:
  - Não há mensagens espúrias (*no creation*): se uma mensagem é recebida pelo processo *q* então ela foi efetivamente transmitida pelo processo *p*
  - Duplicação Finita: se uma mensagem é transmitida um número finito de vezes pelo processo *p*, então ela *não* é recebida infinitas vezes pelo processo *q*
  - A probabilidade de uma mensagem ser recebida é maior que zero *(fair-loss):* próximo slide!

Propriedade *Fair-Loss*

- Propriedade *Fair-Loss:* se uma mensagem é transmitida infinitas vezes pelo processo *p,* e ambos os processos não falham, a mensagem é recebida infinitas vezes processo *q*
- A probabilidade de uma mensagem ser recebida é maior que zero

Um Modelo de Enlace Intermediário

- No livro Guerraoui-Rodrigues entre o enlace fair-loss e o enlace perfeito, apresenta um enlace intermediário (*não é um modelo comum*):
- Canais de Comunicação Teimosos (*Stubborn*)
- Um canal de comunicação *stubborn* é construído sobre um canal de comunicação *fair-loss*
- Tem duas propriedades:
  - *No-Creation (não há mensagens espúrias)*
  - *Stubborn Delivery (“entrega teimosa”)*

Stubborn Delivery

- De acordo com esta propriedade dos canais de comunicação teimosos:
- Se o processo *p* transmite uma mensagem para o processo *q*, e ambos os processos não falham, então o processo *q* recebe esta mensagem infinitas vezes
- Garante que as perdas de mensagens não afetam a comunicação :-)

O Algoritmo do Enlace Teimoso

Algoritmo Enlace Teimoso

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

O Algoritmo do Enlace Teimoso

Algoritmo Enlace Teimoso

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent: conjunto de mensagens transmitidas

TimeDelay: intervalo de retransmissão pré-definido

O Algoritmo do Enlace Teimoso

Algoritmo Enlace Teimoso

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent: conjunto de mensagens transmitidas

TimeDelay: intervalo de retransmissão pré-definido

Init: Sent ← vazio;

Start-Timer(TimeDelay);

O Algoritmo do Enlace Teimoso

Algoritmo Enlace Teimoso

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent: conjunto de mensagens transmitidas

TimeDelay: intervalo de retransmissão pré-definido

Init: Sent ← vazio;

Start-Timer(TimeDelay);

UPON Timeout: for all msg in Sent do send(msg);

Start-Timer(TimeDelay);

O Algoritmo do Enlace Teimoso

Algoritmo Enlace Teimoso

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent: conjunto de mensagens transmitidas

TimeDelay: intervalo de retransmissão pré-definido

Init: Sent ← vazio;

Start-Timer(TimeDelay);

UPON Timeout: for all msg in Sent do send(msg);

Start-Timer(TimeDelay);

UPON there is a new msg to transmit: Sent ← Sent U {msg};

send(msg);

O Algoritmo do Enlace Teimoso

Algoritmo Enlace Teimoso

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */
Var Sent: conjunto de mensagens transmitidas
TimeDelay: intervalo de retransmissão pré-definido
Init: Sent ← vazio;
Start-Timer(TimeDelay);
UPON Timeout: for all msg in Sent do send(msg);
Start-Timer(TimeDelay);
UPON there is a new msg to transmit: Sent ← Sent U {msg};
send(msg);
UPON receive(msg): deliver(msg);

Canais de Comunicação Perfeitos

- Para que ficar entregando a mensagem infinitas vezes? Totalmente desnecessário
- Propriedade importante: só entrega cada mensagem 1 única vez, *No Duplication*
- Além disso: *No Creation*, e
- Entrega Confiável (*Reliable Delivery*): se o processo p transmite uma mensagem para o processo e ambos não falham: o processo q recebe a mensagem *eventually*

A Palavra *Eventually*

- Muito, muito usada em Sistemas Distribuídos
- Significa que não se sabe quando vai acontecer, mas 100% vai acontecer em algum momento
- Diferente de eventualmente em português! :-(
- “Eventualmente vou visitar fulano esta semana”
- “Eventualmente pode chover hoje”
- *Eventually* tem sido traduzida como “em algum momento” ou “a termo” :-(:-0

Algoritmo Enlace Perfeito

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent: conjunto de mensagens transmitidas

Delivered: conjunto de mensagens entregues

TimeDelay: intervalo de retransmissão pré-definido

Init: Sent ← vazio; Delivered ← vazio;

Start-Timer(TimeDelay);

Algoritmo Enlace Perfeito

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent: conjunto de mensagens transmitidas

Delivered: conjunto de mensagens entregues

TimeDelay: intervalo de retransmissão pré-definido

Init: Sent ← vazio; Delivered ← vazio;

Start-Timer(TimeDelay);

UPON Timeout: for all msg in Sent do send(msg);

Start-Timer(TimeDelay);

Algoritmo Enlace Perfeito

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent: conjunto de mensagens transmitidas

Delivered: conjunto de mensagens entregues

TimeDelay: intervalo de retransmissão pré-definido

Init: Sent ← vazio; Delivered ← vazio;

Start-Timer(TimeDelay);

UPON Timeout: for all msg in Sent do send(msg);

Start-Timer(TimeDelay);

UPON there is a new msg to transmit: Sent ← Sent U {msg};

send(msg);

Algoritmo Enlace Perfeito

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent: conjunto de mensagens transmitidas

Delivered: conjunto de mensagens entregues

TimeDelay: intervalo de retransmissão pré-definido

Init: Sent ← vazio; Delivered ← vazio;

Start-Timer(TimeDelay);

UPON Timeout: for all msg in Sent do send(msg);

Start-Timer(TimeDelay);

UPON there is a new msg to transmit: Sent ← Sent U {msg};

send(msg);

UPON receive(msg): if msg not in Delivered then deliver(msg);

Delivered ← Delivered U {msg};

Ficar Retransmitindo Mensagens Infinitamente?

- Na prática, ao invés de retransmitir cada mensagem infinitamente, para garantir o recebimento...
- ... usaria ACK! Confirmação de recebimento
- Mas cuidado: o canal de comunicação é *fair-loss,* perde mensagens, assim o ACK pode não chegar!

Canal de Comunicação Perfeito com ACK

- Podemos implementar com ACKs, sempre lembrando que o ACK pode não chegar!
- Assim retransmite até o ACK chegar
- Pode retransmitir infinitas vezes!
  - Significado teórico: pior caso é infinitas retransmissões
- Nossa estratégia: chegou um ACK? Remove a mensagem confirmada do conjunto Sent
- Quando recebe uma mensagem, sempre manda um ACK, mesmo que já tiver entregue/recebido antes

Algoritmo Enlace Perfeito com ACKs

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent, Delivered, TimeDelay;

Init: Sent ← vazio; Delivered ← vazio;

Start-Timer(TimeDelay);

Algoritmo Enlace Perfeito com ACKs

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent, Delivered, TimeDelay;

Init: Sent ← vazio; Delivered ← vazio;

Start-Timer(TimeDelay);

UPON Timeout: for all msg in Sent do send(msg);

Start-Timer(TimeDelay);

Algoritmo Enlace Perfeito com ACKs

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent, Delivered, TimeDelay;

Init: Sent ← vazio; Delivered ← vazio;

Start-Timer(TimeDelay);

UPON Timeout: for all msg in Sent do send(msg);

Start-Timer(TimeDelay);

UPON there is a new msg to transmit: Sent ← Sent U {msg}; send(msg);

Algoritmo Enlace Perfeito com ACKs

Usa Enlace Fair-Loss /* *Construído sobre enlace fair-loss* */

Var Sent, Delivered, TimeDelay;

Init: Sent ← vazio; Delivered ← vazio;

Start-Timer(TimeDelay);

UPON Timeout: for all msg in Sent do send(msg);

Start-Timer(TimeDelay);

UPON there is a new msg to transmit: Sent ← Sent U {msg}; send(msg);
UPON receive(msg): if msg is an ACK
then remove confirmed msg from Sent;
else if msg not in Delivered then deliver(msg);
Delivered ← Delivered U {msg};
send(ACK for the received msg)

Uma Forma de Enxergar na Prática

- Comunicação sobre canal perfeito: usando TCP
- Comunicação sobre canal *fair-loss:* UDP e/ou IP
- Sim, podemos considerar que o TCP implementa um canal de comunicação perfeito

Obrigado!
Página da disciplina
Sistemas Distribuídos:
www.inf.ufpr.br/elias/sisdis