ACH 2147 — DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO DISTRIBUÍDOS

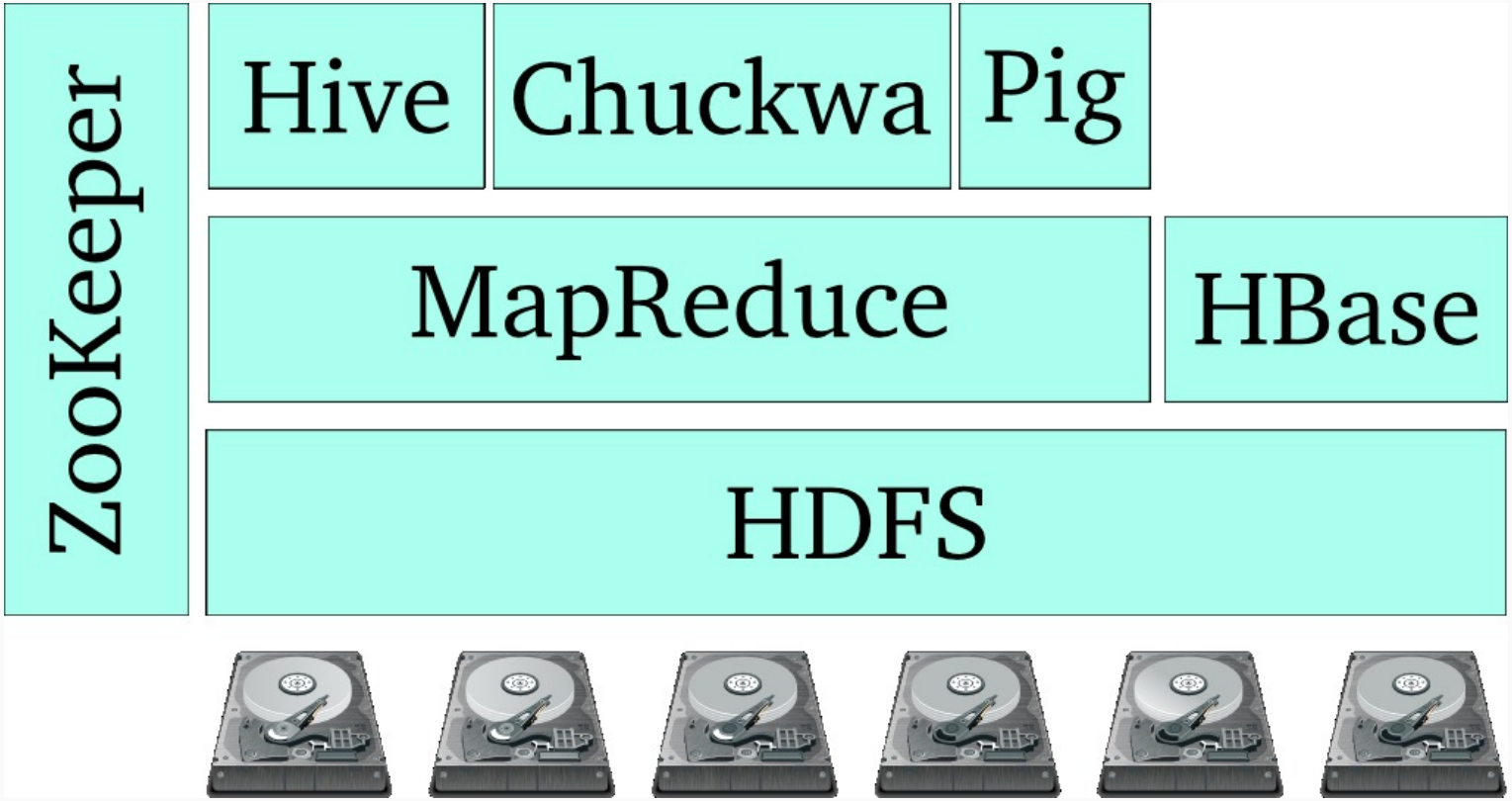
APACHE HADOOP E SPARK

Daniel Cordeiro

6 e 8 de junho de 2017

Escola de Artes, Ciências e Humanidades | EACH | USP

A PILHA DE SOFTWARE DO HADOOP



PROJETOS APACHE RELACIONADOS

Ambari gerenciamento de *clusters* Hadoop

Avro seriação de dados e chamada a procedimentos remotos (*Remote Procedure Call*)

Cassandra banco de dados NoSQL, tuplas <chave,valor>

Chukwa monitoramento e coleta de dados de sistemas distribuídos

HBase banco de dados não-relacional distribuído e escalável (baseado no Google Bigtable)

Hive infraestrutura de *data warehouse* (relacional, SQL-like)

PROJETOS APACHE RELACIONADOS

Mahout biblioteca para *machine learning* e *data mining*

Pig plataforma de análise de dados e linguagem de fluxo de dados (Pig Latin)

Spark plataforma de computação para aplicações que reusam dados em processos paralelos

Tez plataforma de computação para processamento de fluxos de dados

ZooKeeper coordenação de serviços distribuídos (configurações, nomes, sincronização, etc.)

Estender o modelo MapReduce para apoiar duas classes comuns de aplicações de análise de dados:

- Algoritmos iterativos (aprendizado de máquina, grafos, etc.)
- Algoritmos interativos (mineração de dados)

MOTIVAÇÃO

O modelo MapReduce é um modelo de *data flow* acíclico, com armazenamento estável em todas as etapas.

- isso é ótimo para recuperação automática de falhas e balanceamento de carga
- mas péssimo para aplicações que reutilizam um mesmo conjunto de dados

SOLUÇÃO

Uso de Resilient Distributed Datasets (RDDs):

- permite manter os dados em memória para aumentar eficiência
- garante as boas propriedades de MapReduce
  - tolerância a falhas, localidade e escalabilidade
- é um ótimo modelo abstrato para uma grande variedade de aplicações

Resilient Distributed Datasets (RDDs)

- Coleções imutáveis de objetos distribuídas em vários nós
- Criada com *transformações* paralelas (map, filter, groupby) em dados armazenados em disco
- Permite a execução eficiente de ações como count, reduce, collect, etc.
- As operações mantém a "linhagem" dos dados (permite tolerância a falhas por saber qual o conjunto de operações que geraram os dados)

ESCALONAMENTO DAS TAREFAS

- DAGs à lá Dryad
- *Pipelines* em um mesmo *stage*
- Distribuição de tarefas ciente de cache
- Ciente das partições para evitar *shuffle*

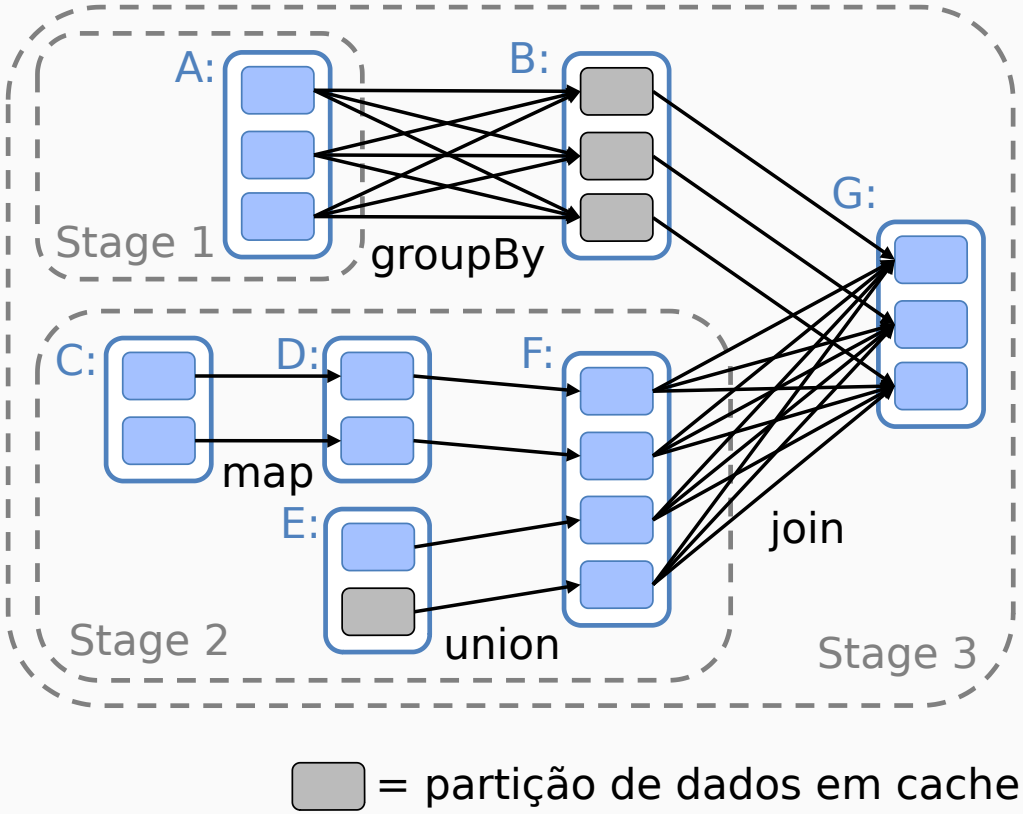


Figura: Para executar uma ação no RDD G, o Spark calcula as dependências e escalona a execução das transformações em cada estágio. Nesse caso, o RDD de saída do estágio 1 já está na memória, então o Spark executa o estágio 2 e depois o 3.

EXEMPLO: FILTRAGEM DE LOGS

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
```

Capaz de processar 1 TB de dados em 5–7 segundos!

Transformações (definem um novo RDD)

map · filter · sample · groupByKey · reduceByKey · sortByKey · flatMap · union · join · cogroup · cross · mapValues

Ações (devolvem um resultado ao *driver*)

collect · reduce · count · save · lookupKey

EXEMPLO: REGRESSÃO LOGÍSTICA

O Spark oferece API em Python, Scala e Java.

```
points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # separating plane
for i in range(ITERATIONS):
  gradient = points.map(lambda p:
    (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
  ).reduce(lambda a, b: a + b)
  w -= gradient
print "Final separating plane: %s" % w
```

EXEMPLO: REGRESSÃO LOGÍSTICA

O Spark oferece API em Python, Scala e Java.

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating plane: " + w)
```

EXEMPLO: REGRESSÃO LOGÍSTICA

O Spark oferece API em Python, Scala e Java.

```
class ComputeGradient extends Function<DataPoint, Vector> {
  private Vector w;
  ComputeGradient(Vector w) { this.w = w; }
  public Vector call(DataPoint p) {
    return p.x.times(p.y * (1 / (1 + Math.exp(w.dot(p.x))) - 1));
  }
}

JavaRDD<DataPoint> points = spark.textFile(...).map(new ParsePoint()).cache();
Vector w = Vector.random(D); // current separating plane
for (int i = 0; i < ITERATIONS; i++) {
  Vector gradient = points.map(new ComputeGradient(w)).reduce(new AddVectors());
  w = w.subtract(gradient);
}
System.out.println("Final separating plane: " + w);
```

REFERÊNCIAS

- Tom White. **Hadoop: The Definitive Guide**. Yahoo Press. ISBN: 9781449311520
- Chuck Lam. **Hadoop in Action**. Manning Publications. ISBN: 9781935182191
- Alfredo Goldman *et al.* Apache Hadoop: conceitos teóricos e práticos, evolução e novas possibilidades. Em: XXXI Jornadas de Atualização em Informática. Sociedade Brasileira de Computação, 2012. `http://www.lbd.dcc.ufmg.br/colecoes/jai/2012/003.pdf`

Coordenação

SINCRONIZAÇÃO DE RELÓGIOS

- relógios físicos
- relógios lógicos
- relógios vetoriais

Problema

Algumas vezes precisamos saber a hora exata e não apenas uma ordenação de eventos.

Universal Coordinated Time (UTC):

- baseado no número de transições por segundo do átomo de césio 133 (bastante preciso)
- atualmente, o tempo é medido como a média de cerca de 50 relógios de césio espalhados pelo mundo
- introduz um *segundo bissexto* de tempos em tempos para compensar o fato de que os dias estão se tornando maiores

Nota:

O valor do UTC é enviado via *broadcast* por satélite e por ondas curtas de rádio. Satélites tem um acurácia de ± 0.5 ms.

Precisão

O objetivo é tentar fazer com que o desvio entre dois relógios em quaisquer duas máquinas fique dentro de um limite especificado, conhecido como a precisão π:

$$\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$$

onde $C_p(t)$ é o horário do relógio computado para a máquina p no horário UTC t.

Acurácia

No caso da acurácia, queremos manter o relógio limitado a um valor α:

$$\forall t, \forall p : |C_p(t) - t| \leq \alpha$$

Sincronização

Sincronização interna: manter a precisão dos relógios

Sincronização externa: manter a acurácia dos relógios

Especificação dos relógios

- Todo relógio tem especificado sua taxa máxima de desvio do relógio ρ.
- $F(t)$: frequência do oscilador do relógio do hardware no tempo t
- F: frequência (constante) do relógio ideal:

$$\forall t : (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho)$$

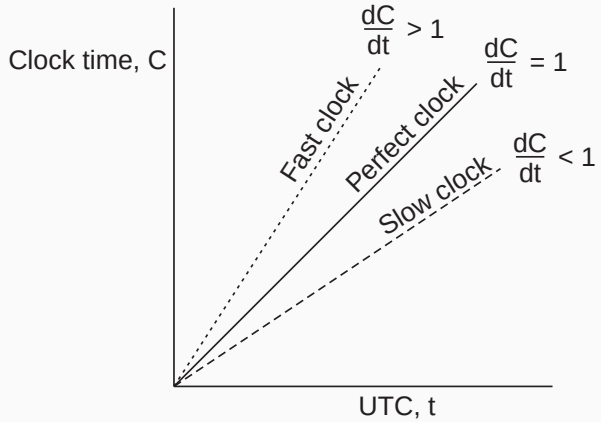
Observação

Interrupções de hardware acoplam um relógio de software a um relógio de hardware, que também tem sua taxa de desvio:

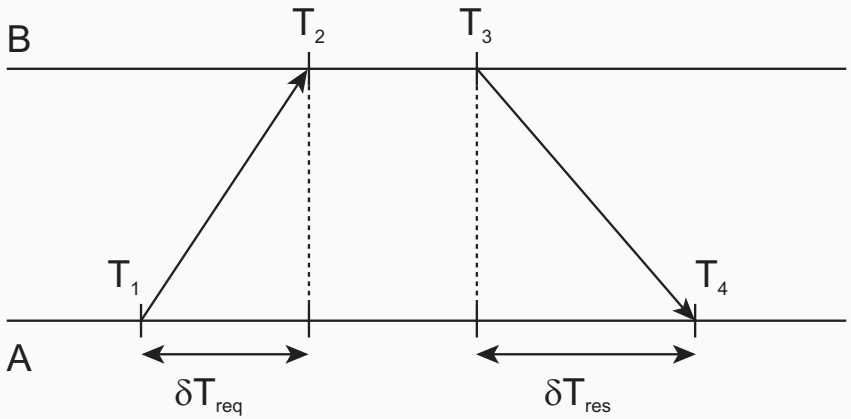
$$C_p(t) = \frac{1}{F} \int_0^t F(t)dt \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F}$$

$$\Rightarrow \forall t : 1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho$$

Relógios rápidos, perfeitos e lentos



Recuperação do horário atual de um servidor



Cálculo da diferença relativa θ e o atraso δ

Assumindo que: $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

$$\theta = T_3 + \big((T_2 - T_1) + (T_4 - T_3)\big)/2 - T_4 = \big((T_2 - T_1) + (T_3 - T_4)\big)/2$$

$$\delta = \big((T_4 - T_1) - (T_3 - T_2)\big)/2$$

Network Time Protocol

Colete oito pares (θ, δ) e escolha os θ cujos atrasos δ são minimais.

Sincronização externa

Cada máquina pede a um *servidor de hora* a hora certa pelo menos uma vez a cada $\delta/(2\rho)$ (Network Time Protocol)

OK, mas...

você ainda precisa de uma maneira precisa de medir o *round trip delay*, incluindo o tratamento da interrupção e o processamento das mensagens.

Sincronização interna

Permita o servidor de hora sonde todas as máquinas periodicamente, calcule uma média e informe cada máquina como ela deve ajustar o seu horário relativo ao seu horário atual.

Nota:

Você provavelmente terá todas as máquinas em sincronia. Você nem precisa propagar o horário UTC.

É fundamental

saber que atrasar o relógio nunca é permitido. Você deve fazer ajustes suaves.

Relógios lógicos

O que importa na maior parte dos sistemas distribuídos não é fazer com que todos os processos concordem exatamente com o horário, mas sim fazer com que eles concordem com a ordem em que os eventos ocorreram. Ou seja, precisamos de uma noção de ordem entre os eventos.

A relação “aconteceu-antes” (*happened-before*)

- se a e b são dois eventos de um mesmo processo e a ocorreu antes de b, então $a \rightarrow b$

A relação “aconteceu-antes” (*happened-before*)

- se a e b são dois eventos de um mesmo processo e a ocorreu antes de b, então $a \rightarrow b$
- se a for o evento de envio de uma mensagem e b for o evento de recebimento desta mesma mensagem, então $a \rightarrow b$

A relação “aconteceu-antes” (*happened-before*)

- se a e b são dois eventos de um mesmo processo e a ocorreu antes de b, então $a \rightarrow b$
- se a for o evento de envio de uma mensagem e b for o evento de recebimento desta mesma mensagem, então $a \rightarrow b$
- se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$

A relação “aconteceu-antes” (*happened-before*)

- se a e b são dois eventos de um mesmo processo e a ocorreu antes de b, então $a \rightarrow b$
- se a for o evento de envio de uma mensagem e b for o evento de recebimento desta mesma mensagem, então $a \rightarrow b$
- se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$

A relação “aconteceu-antes” (*happened-before*)

- se a e b são dois eventos de um mesmo processo e a ocorreu antes de b, então $a \rightarrow b$
- se a for o evento de envio de uma mensagem e b for o evento de recebimento desta mesma mensagem, então $a \rightarrow b$
- se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$

Nota:

Isso introduz uma noção de ordem parcial dos eventos em um sistema com processos executando concorrentemente.

Problema

Como fazemos para manter uma visão global do comportamento do sistema que seja consistente com a relação aconteceu-antes?

Problema

Como fazemos para manter uma visão global do comportamento do sistema que seja consistente com a relação aconteceu-antes?

Solução

Associar um *timestamp* $C(e)$ a cada evento e tal que:

P1 se a e b são dois eventos no mesmo processo e $a \rightarrow b$, então é obrigatório que $C(a) < C(b)$

P2 se a corresponder ao envio de uma mensagem m e b ao recebimento desta mensagem, então também é válido que $C(a) < C(b)$

Problema

Como fazemos para manter uma visão global do comportamento do sistema que seja consistente com a relação aconteceu-antes?

Solução

Associar um *timestamp* $C(e)$ a cada evento e tal que:

P1 se a e b são dois eventos no mesmo processo e $a \rightarrow b$, então é obrigatório que $C(a) < C(b)$

P2 se a corresponder ao envio de uma mensagem m e b ao recebimento desta mensagem, então também é válido que $C(a) < C(b)$

Outro problema

Como associar um *timestamp* a um evento quando não há um relógio global? Solução: manter um conjunto de relógios lógicos consistentes, um para cada processo

Solução

Cada processo P_i mantém um contador C_i local e o ajusta de acordo com as seguintes regras:

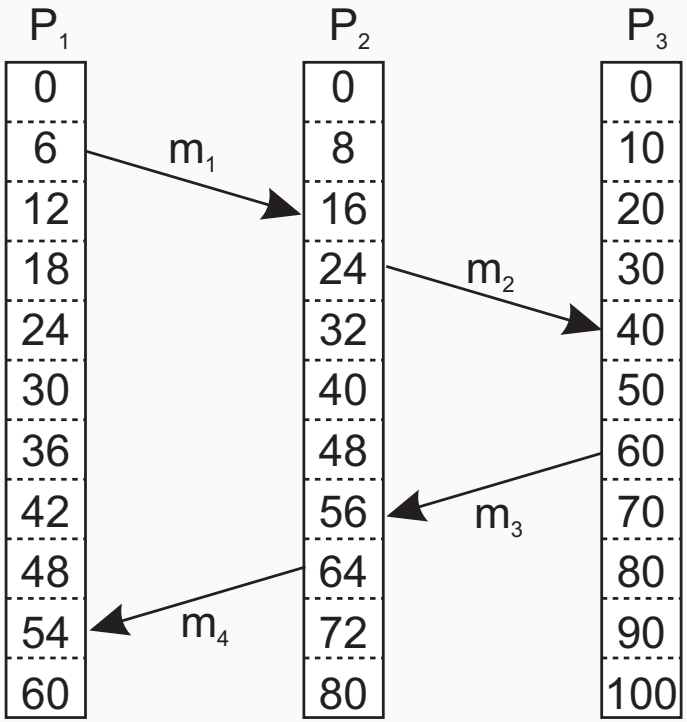
1. para quaisquer dois **eventos sucessivos** que ocorrer em P_i, C_i é incrementado em 1
2. toda vez que uma mensagem m for enviada por um processo P_i, a mensagem deve receber um *timestamp* $ts(m) = C_i$
3. sempre que uma mensagem m for recebida por um processo P_j, P_j ajustará seu contador local C_j para $\max\{C_j, ts(m)\}$ e executará o passo 1 antes de repassar m para a aplicação

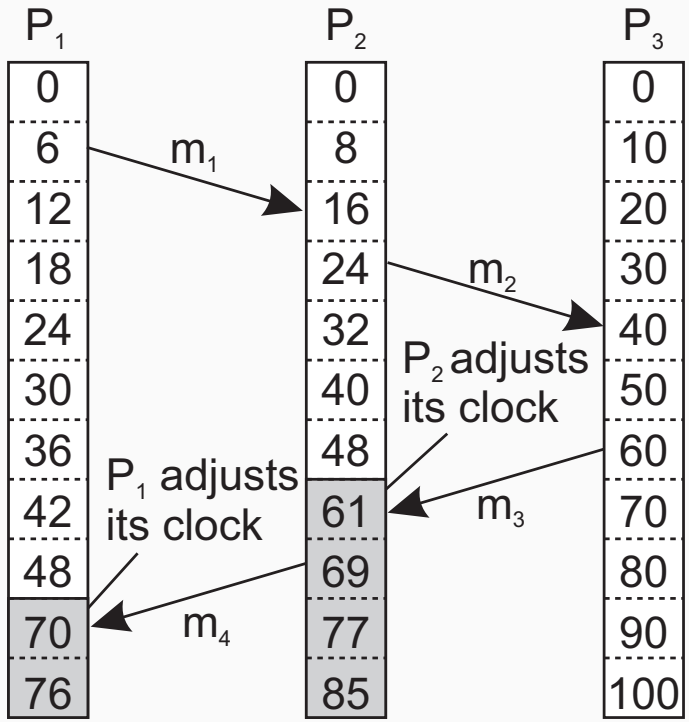
Observações:

- a propriedade **P1** é satisfeita por (1); propriedade **P2** por (2) e (3)
- ainda assim pode acontecer de dois eventos ocorrerem ao mesmo tempo. Desempate usando os IDs dos processos.

RELÓGIO LÓGICO DE LAMPORT – EXEMPLO

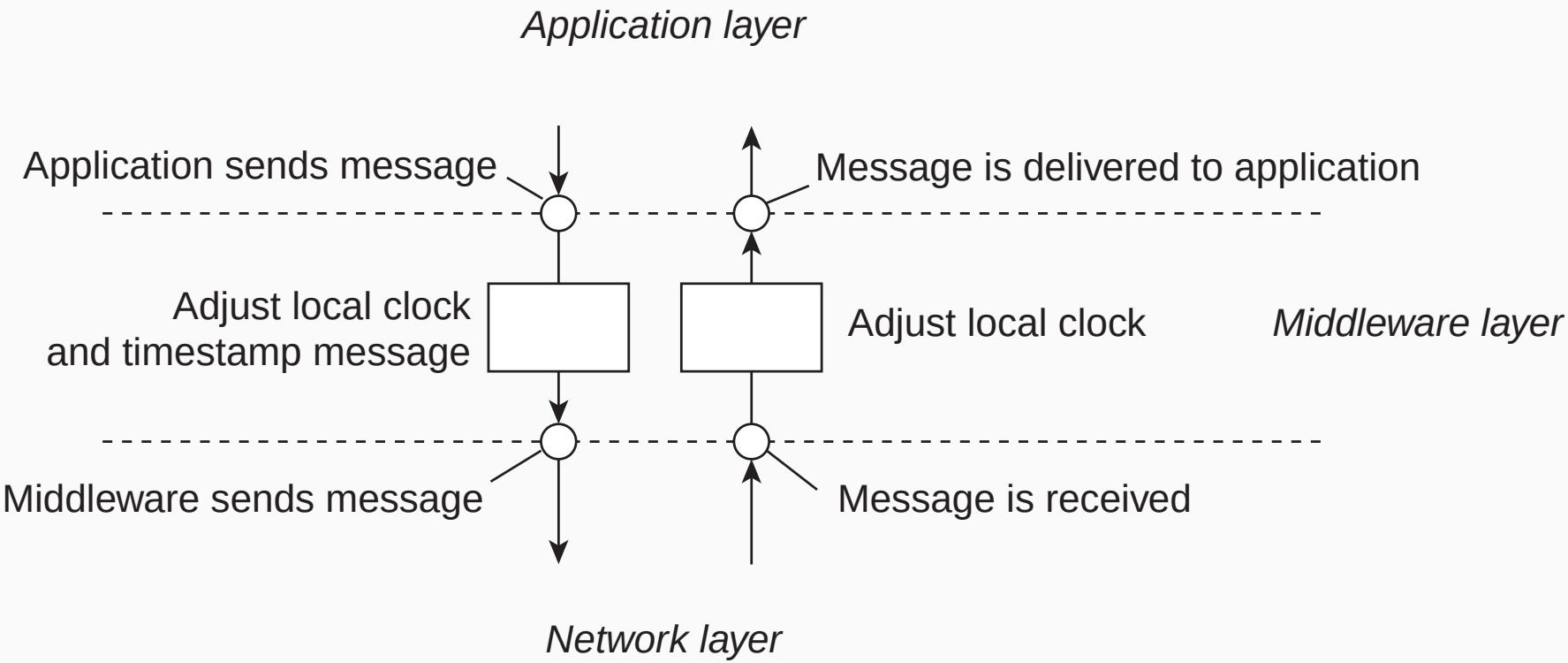
Considere três processos com contadores de eventos funcionando a velocidades diferentes.





Nota

Os ajustes ocorrem na camada do *middleware*

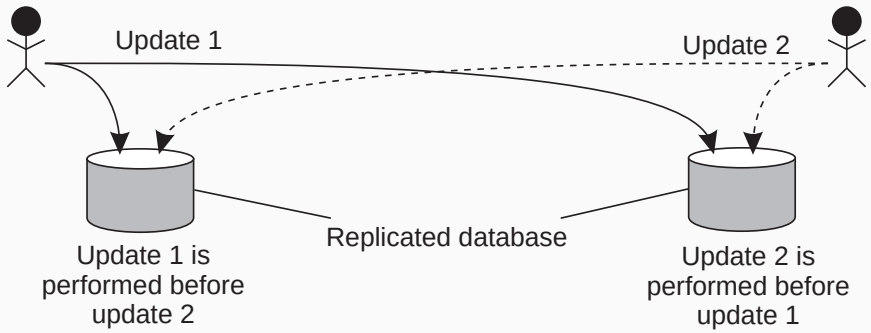


EXEMPLO: MULTICAST COM ORDEM TOTAL

Problema

Alguma vezes precisamos garantir que atualizações concorrentes em um banco de dados replicado sejam vistos por todos como se tivessem ocorrido na mesma ordem.

- P_1 adiciona R$ 100 a uma conta (valor inicial: R$ 1000)
- P_2 incrementa a conta em 1%
- Há duas réplicas



Resultado

Na ausência de sincronização correta,
réplica #1 ← R$ 1111, enquanto que na réplica #2 ← R$ 1110.

Solução

- processo P_i envia uma mensagem com timestamp m_i para todos os outros. A mensagem é colocada em sua fila local $queue_i$.
- toda mensagem que chegar em P_j é colocada na fila $queue_j$ priorizada pelo seu timestamp e confirmada (*acknowledged*) por todos os outros processos

P_j repassa a mensagem m_i para a sua aplicação somente se:

(1) m_i estiver na cabeça da fila $queue_j$

(2) para todo processo P_k, existe uma mensagem m_k na $queue_j$ com um *timestamp* maior.

Nota

Assumimos que a comunicação é confiável e que a ordem FIFO é respeitada.

O ALGORITMO DE MULTICAST FUNCIONA?

Observe que:

- se uma mensagem *m* ficar ponta em um servidor *S*, *m* foi recebido por todos os outros servidores (que enviaram ACKs dizendo que *m* foi recebido)
- se *n* é uma mensagem originada no mesmo lugar que *m* e for enviada antes de *m*, então todos receberão *n* antes de *m* e *n* ficará no topo da fila antes de *m*
- se *n* for originada em outro lugar, é um pouco mais complicado. Pode ser que *m* e *n* cheguem em ordem diferente nos servidores, mas é certeza de que antes de tirar um deles da fila, ele terá que receber os ACKs de todos os outros servidores, o que permitirá comparar os valores dos relógios e entregar para as mensagens na ordem total dos relógios

RELÓGIO DE LAMPORT PARA EXCLUSÃO MÚTUA

```
class Process:
  def __init__(self, chan):
    self.queue      = []                    # The request queue
    self.clock      = 0                     # The current logical clock

  def requestToEnter(self):
    self.clock = self.clock + 1                          # Increment clock value
    self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
    self.cleanupQ()                                      # Sort the queue
    self.chan.sendTo(self.otherProcs, (self.clock,self.procID,ENTER)) # Send request

  def allowToEnter(self, requester):
    self.clock = self.clock + 1                          # Increment clock value
    self.chan.sendTo([requester], (self.clock,self.procID,ALLOW)) # Permit other

  def release(self):
    tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ALLOWs
    self.queue = tmp                                     # and copy to new queue
    self.clock = self.clock + 1                          # Increment clock value
    self.chan.sendTo(self.otherProcs, (self.clock,self.procID,RELEASE)) # Release

  def allowedToEnter(self):
    commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
    return (self.queue[0][1]==self.procID and len(self.otherProcs)==len(commProcs))
```

RELÓGIO DE LAMPORT PARA EXCLUSÃO MÚTUA

```
def receive(self):
  msg = self.chan.recvFrom(self.otherProcs)[1]  # Pick up any message
  self.clock = max(self.clock, msg[0])          # Adjust clock value...
  self.clock = self.clock + 1                   # ...and increment
  if msg[2] == ENTER:
    self.queue.append(msg)                      # Append an ENTER request
    self.allowToEnter(msg[1])                   # and unconditionally allow
  elif msg[2] == ALLOW:
    self.queue.append(msg)                      # Append an ALLOW
  elif msg[2] == RELEASE:
    del(self.queue[0])                          # Just remove first message
  self.cleanupQ()                               # And sort and cleanup
```

Analogia com multicast de ordem total

- No multicast de ordem total, todos os processos construíam fila idênticas, entregando as mensagens na mesma ordem
- Exclusão mútua implica em concordar sobre a ordem em que os processos devem ter sua entrada permitida na seção crítica