SistemasDistribuidos - Lista3

Aluno: Marcos Seefelder de Assis Araujo

19-07-2016

1.

- Processo que chama e processo que executa função estão em máquinas diferentes, acarretando em espaços de endereçamento e ambiente computacional distintos;
- Representação de dados diferentes (Little Endian, Big Endian);
- Falhas de rede e das máquinas envolvidas.

Extra: Little Endian vs. Big Endian

Sistemas representam vetores de bytes de maneiras diferentes. Considerando que temos uma *word* (32 *bits*) representada em hexadecimal por **0A0B0C0D**:

- Big Endian (**MSB primeiro**): Vai representar na memória (m[]) como m[0] = 0A; m[1] = 0B; m[2] = 0C; m[3] = 0D;
- Little Endian (**LSB primeiro**): Vai representar na memória (m[]) como m[0] = 0D; m[1] = 0C; m[2] = 0B; m[3] = 0A;

2.

1. Cliente faz chamada de função para servidor de RPC e espera (bloqueante) mensagem de aceitação;
2. Servidor de RPC recebe chamada de função e envia mensagem de aceitação para cliente. Servidor passa a executar procedimento local...;
3. Cliente recebe aceitação, a chamada de função retorna mas ainda não há resultado. Cliente continua sua execução...;
4. Ao terminar o procedimento, servidor envia resultado ao cliente via RPC;
5. Mensagem de resultado gera uma interrupção no cliente, que por sua vez envia o reconhecimento de que recebeu a mensagem ao servidor.

Extra: *Marshalling*, *Unmarshalling* e *Stubs*

Para evitar problemas por divergências nas representações de dados entre os participantes de um sistema de RPC, é feito o *marshalling* de mensagens a serem enviadas, convertendo a mensagem para uma representção de comum acordo entre os participantes. Ao receber uma mensagem, é feito o *unmarshalling* da mesma.

Tanto o *marshalling* quanto o *unmarshalling* são feitos pelos *stubs*, que são trechos de código fornecidos pelas biblioecas de RPC e tornam o envio e recebimento de chamadas e retornos de RPC transparentes aos usuários.

3.

A sincronização de relógio resolve o problema da *odenação global de eventos* para as máquinas cujos relógios foram sincronizados. Um exemplo de situação gerada por esse problema de ordenação em Sistemas Distribuídos é:

- Dois usuários compartilam um diretório no Dropbox;
- O **usuário 1** envia a versão **v1** de um documento e seu computador marca *12:00* no momento do envio;
- O **usuário 2** envia a versão **v2** do mesmo documento *5 minutos* depois, mas o relógio do computador dele está *1 hora* atrasado em marca *11:05* na hora do envio;
- A versão mais recente para o Dropbox fica como **v1**.

Caso os relógios dos usuários estivessem sincronizados no exemplo anterior, o problema não ocorreria.

4.

A *sincronização de relógios* tem **dois problemas principais**: Qual é a hora certa e como fazer a sincronização.

O problema da hora certa é resolvido utilizando padrões internacionais e protocolos para definir uma hora certa para todos, sendo UTC (Universal Time Coordinated) o padrão utilizado atualmente. O mesmo é baseado em dois componentes: o TAI (International Atomic Time), e combina os tempos de cerca de 400 relógios atômicos de alta precisão espalhados pelo mundo, e o UT1 que é referente à rotação da Terra.

Já o método de sincronização atual para computadores é o Network Time Protocol (NTC) que utiliza UTC como referência. O sistema funciona como uma hierarquia de servidores que vão da maior pra menor precisão. Nesse sistema, clientes solicitam a hora periodicamente a três ou mais servidores e determina sua hora utilizando uma média com filtros (para a remoção de *outliers*) dos valores obtidos. A hora não é alterada imediatamente, para evitar saltos temporais e a volta no tempo. O que ocorre é que ataxa de progressão do relógio é ajustada para que o relógio local se ajuste suavemente para UTC.

5.

Considerando dois computadores A e B, B quer usar o relógio de A como referência para a sincronização:

1. B envia uma solicitação em $T1$ (tempo registrado em B) para A, guardadndo o valor de $T1$;
2. A recebe a solicitação em $T2$, processa a mesma e envia a resposta em $T3$ para B, mandando na mensagem os valores de $T2$ e $T3$ (tempos registrados em A);
3. B recebe a resposta em $T4$ (tempo registrado em B);
4. Assumindo que o retardo de ida é igual ao de volta, B calcula o mesmo como $d = ((T_4 - T_1) - (T_3 - T_2))/2$ e ajusta seu relógio para $T_3 + d$.

Dessa maneira, B é sincronizado a A.

6.

O Network Time Protocol (NTP) funciona para sincronizar relógios com o padrão UTC e é estrutuado com uma hierarquia de servidores e clientes. No topo da hierarquia estão os computadores com relógios de maior precisão. Descendo na hierarquia, a precisão diminui.

Um determinado computador em um nível da hierarquia solicita a hora periódicamente para três ou mais servidores de uma hierarquia superior. Ele faz então uma filtragem dos resultados, para remover *outliers*, tira uma média e ajusta sua taxa de progressão para se ajustar suavemente ao horário padrão.

7.

1. Relações $x \to y$:
   - $a \to r$
   - $h \to b$
   - $i \to n$
   - $s \to j$
   - $c \to o$
   - $d \to t$
   - $k \to e$
   - $p \to f$
   - $u \to l$
   - $g \to q$
2. Uma relação $x \parallel y$:
   - $p \parallel u$
3. Relações no conjunto $E = b, k, n, u$:
   - $b \parallel k$
   - $b \parallel n$
   - $b \to u$: $b \to c$, $c \to d$, $d \to t$, $t \to u$
   - $k \parallel n$
   - $k \parallel u$
   - $n \parallel u$
4. Relógios de Lamport para cada evento:
   - L(a) = 1
   - L(b) = 2
   - L(c) = 3
   - L(d) = 4
   - L(e) = 6
   - L(f) = 10
   - L(g) = 11
   - L(h) = 1
   - L(i) = 3
   - L(j) = 4
   - L(k) = 5
   - L(l) = 8
   - L(m) = 1
   - L(n) = 4
   - L(o) = 8
   - L(p) = 9
   - L(q) = 12
   - L(r) = 2
   - L(s) = 3
   - L(t) = 5
   - L(u) = 7
5. Relógios de vetor:
   - V(a) = [1,0,0,0]
   - V(b) = [2,1,0,0]
   - V(c) = [3,1,0,0]
   - V(d) = [4,1,0,0]
   - V(e) = [5,5,1,2]
   - V(f) = [6,5,5,4]
   - V(g) = [7,5,5,4]
   - V(h) = [0,1,0,0]
   - V(i) = [0,3,1,0]

- V(j) = [1,4,1,2]
- V(k) = [1,5,1,2]
- V(l) = [4,6,1,5]
- V(m) = [0,0,1,0]
- V(n) = [0,3,2,0]
- V(o) = [4,3,4,4]
- V(p) = [4,3,5,4]
- V(q) = [7,5,6,4]
- V(r) = [1,0,0,1]
- V(s) = [1,0,0,2]
- V(t) = [4,1,0,3]
- V(u) = [4,1,0,5]

6. Exemplo onde para eventos x e y, $L(x) < L(y)$ e $V(y) < V(x)$: Não existe, pois o relógio de vetor não discorda de lamport na ordenação.
7. Se $L(x) < L(y)$: Nada pode ser concluído;
8. Se $V(x) < V(y)$: Podemos concluir que $x \rightarrow y$

8.

Questão feita à mão no final do documento.

9.

Vantagem: Para um sistema com filas FIFO e sem prioridade o algoritmo é **correto** (garante o acesso exclusivo) e **justo** com a troca de apenas três mensagens por acesso à região crítica;

Desvantagem: O algoritmo tem um ponto unico de falha, que é o coordenador da exclusão mútua. Caso o mesmo falhe de alguma maneira, o sistema falha junto.

10.

É interessante que um nó conheça seus dois próximos vizinhos pois no caso de falha de um nó o anel não será quebrado, pois o nó que se conectava com o que falhou só precisa pular o mesmo. Para o caso de falhas de mais vizinhos, seria vantajoso conhecer mais do uqe dois vizinhos.

11.

ACID é a sigla que representa o conjunto de propriedades que um sistema transacional deve apresentar. O significado de suas letras é:

"A" de *Atomicity*: Uma transação ou é aplicada por inteiro ou é abortada por inteiro sem modificar o estado global;

"C" de *Consistency*: Toda transação preserva as *propriedades* do estado global;

"I" de *Isolation*: Ao *ler ou escrever* dados, transação executa como se fosse a única no sistema (**transações ocorrem uma depois da outra**);

"D" de *Durability*: Ao concluir uma transação (*commit*), sistema passa a ter um novo estado global que permanece (independente de interferências externas, como falta de energia ou falhas de equipamentos).

12.

Com a implementação pode ocorrer um *deadlock*. Imaginemos que dois clientes C_1 e C_2 querem fazer as operações `transferencia(a, b, v)` e `transferencia(a, b, v)` respectivamente:

- C_1 faz `acquire(a)`e passa por `se (retirada(a, v) >= 0)`;
- Enquanto isso C_2 faz `acquire(b)` e passa por `se (retirada(b, v) >= 0)`;
- C_1 vai ficar esperando pelo `acquire(b)` e C_2 vai ficar esperando pelo `acquire(a)`, para sempre.

Eu resolveria o código fazendo os dois `acquire()` juntos, usando alguma ordenação global:

```
transferencia(c1, c2, v) {
    acquire(min(c1, c2))
    acquire(max(c1, c2))
    se (retirada(c1, v) >= 0)
        deposito(c2, v)
        release(c1)
        release(c2)
        retorna 0
    release(c1)
    release(c2)
    retorna -1
}
```

13.

Two Phase Locking (2PL):

Mecanismo que serve para controle de concorrência e para a garantia de atomicidade. Faz o uso de **dois tipos de lock para cada objeto**, *read lock* (permitem outros *read locks* simultâneos, mas não um *write lock*) e *write lock* (não permite nenhum tipo de *lock* simultâneo).

Como o nome sugere, são realizadas duas fases:

1. *Expanding*, na qual os *locks* são adquiridos
2. *Shrinking*, na qual os *locks* são liberados

E tem duas variações:

- Strict Two Phase Locking: Na qual na fase de *Shrinking* os *write locks* são liberados apenas no final da transação, mas os read locks podem ser liberados no decorrer da mesma;
- Strong Strict Two Phase Locking: Na qual na fase de *Shrinking*, tando os *read* quanto os *write locks* são liberados apenas no final da transação.

14.

Two Phase Commit (2PC): O algoritmo de 2PL não serve para casos distribuídos, pois considera que o estado global está "centralizado". O 2PC é um protocolo usado para fazer **Commit atômico em sistemas de estado global distribuído**.

O protocolo tem duas fases:

1. Preparar e votar:
    - Coordenador que quer realizar transação (**coordenador**) envia as subtransações adequadas aos demais processos (**participantes**);
    - Cada processo executa 2PL para adquirir os *locks* necessários para a subtransação;
    - Cada processo responde ao coordenador se pode ou não executar a transação

2. Executar:
   - Coordenador recebe todas as respostas e envia:
     - **commit** se todas forem positivas
     - **abort** se houver alguma negativa
   - Participantes recebem a mensagem:
     - caso seja **commit**, efetuam a transação, liberam os *locks* e retornam um OK
     - caso seja **abort**, abortam a transação, liberam os *locks* e retornam um ok

15.

O protocolo 2PC **não evita *deadlocks***, basta imaginar um caso em que ocorra a dependência cíclica dos *locks* levando os participantes a não poderem votar por estarem aguardando *locks* (ex.: `transf(a, b, v)` e `transf(b, a, v)`).

Para lidar com *deadlocks* é definido um *timeout* para a espera por *locks*, que se estourado gera uma resposta negativa ao coordenador. Um coordenaor tenta então realiza a transação repetidamente, porém também é imposto um limite à essa repetição a fim de evitar *livelocks*.

16.

1. Se participante falha em INIT: O coordenador dá *timeout* esperando a resposta e envia um **abort**;
2. Se participante falha em READY: Transação continua normalmente. Ao se recuperar do erro, participante decobre se foi **commit** ou **abort** e efetua ou não a transação;
3. Se coordenador falha em WAIT: Caso algum outro participante já tenha recebido a ordem de **commit** ou **abort**, participantes copiam. Se estiverem todos em estado READY, não tem como decidir o que fazer e aguardam de forma bloqueante, eperando a recuperação do coordenador.

17.

read-write: ocorre quando a leitura e escrita no mesmo dado ocorrem em instâncias diferentes em processos diferentes e a leitura retorna um valor que já está desatualizado devido À escrita recente;

write-write: dois ou mais processos tentam escrever no mesmo dado ao mesmo tempo e o mesmo fica com um valor diferente nas instâncias sendo indefinido qual o valor certo.

18.

- **1**: P2: R(x,0) → P1: W(x,1) → P2: R(x,1)
- **3**: P1: W(x,1) → P3: R(x,1) → P2: W(x,2) → P3: R(x,2)
- **4**: P2: W(x,2) → P3: R(x,2) → P1: W(x,1) → P3: R(x,1)
- **6**: P4 (tudo) → P1 (tudo) → P3 (tudo) → P2 (tudo)

19.

Confiabilidade (*reliability*) diz respeito à **duração** de tempo pela qual um sistema funciona antes de falhar. **Disponibilidade** (*availability*) diz respeito à **fração** de tempo pela qual sistema está operacional.

Um sistema pode ter um MTTF (*Mean Time To Failure*) de 10 anos (alta confiabilidade) e um MTTR (*Mean Time To Repair*) de 8 anos (baixa disponibilidade).

20.

Para um componente, temos disponibilidade de aproximadamente 99.85%. Utilizando a fórmula

$p_k = 1 - (1 - p)^k$

onde p_k é a disponibilidade resultante de um sistema com k componentes de disponibilidade p, se utilizarmos 2 componentes, obtemos uma disponibilidade de aproximadamente 99,9998%. Portanto, **2 componentes são necessários**.

21.

1. Depende. Se não ocorrer nenhuma falha na mesma coluna, para cada nível haverá sempre uma maioria funcionando e o resultado será correto. Porém, se ocorrer de duas das falhas ocorrerem na mesma coluna (dois votadores ou dois componentes), o sistema falha.
2. O sistema falha, pois no próximo nível apenas um componente funcionará corretamente e o resultado no nível não terá maioria.

22.

Pois as mesmas são imprevisíveis e não são fáceis de se detectar. Uma falha bizantina pode passar despercebida, pois os resultados continuam sendo gerados, porém de maneira errada, podendo tornar incorreto o funcionamento do sistema do qual faz parte. Já uma *crash failure*, o componente que falhou para de interferir no sistema, pois simplesmente para de funcionar.

23.

Se um particiante recebe uma mensagem x do coordenador e uma mensagem y do bizantino, ele não poderá chegar a uma conclusão por maioria simples. Portanto, não ocorre um consenso entre esse participante e o coordenador (os não-bizantinos).