Sistemas Distribuidos - Lista 3

Bernardo Dornellas Cysneiros Gomes de Amorim

Questão 1

1. **Representação dos Dados**: Os dados podem ter de ser representados de forma diferentes por diversos motivos, principalmente por conta de linguagens de programação diferente, mas também pode ser por conta de plataformas de hardware diferentes (Little Endian vs Big Endian)
2. **Problemas na Rede**: Uma conexão pode cair no meio de uma chamada RPC. Isso pode ser pior ainda se a chamada tiver *efeitos colaterais*, pois não é possivel simplesmente tentar novamente a chamada, visto que isto pode acarretar em efeitos duplicados (não tem como saber se o servidor recebeu ou não o pedido).

Questão 2

1. O **cliente** faz uma requisição contendo todos os dados necessários para executar a função.
2. O **servidor** responde com um *ack*, informando que recebeu o pedido. Neste momento o **cliente** retoma o controle da sua execução e o **servidor** começa a executar a função.
3. O **servidor** ao terminar o cálculo da função, envia uma mensagem informando o término do cálculo. Essa mensagem tem de criar uma interrupção no cliente, seja por *signal handlers* ou por uma thread no **cliente** embutida na biblioteca de RPC. Só assim o **cliente** poderá decidir o que fazer com o resultado.
4. O **cliente** responde com um *ack*, informando que recebeu o resultado.

Questão 3

Entender a ordem em que eventos aconteceram. Por exemplo, num banco de dados distribuido com réplicas dois clientes inserem numa tabela, qual aconteceu antes?

Questão 4

Nenhum relógio oscila exatamente igual a outro, o que significa que se eles estão marcando a mesma hora num determinado instante eles vão (quase que certamente) estar marcando horários diferentes um certo tempo depois. (Isso sem nem levar em consideração a relatividade).

Além disso, ao sincronizar relógios é necessário alguma "troca de mensagem", a informação tem que sair do relógio de referência e chegar nos outros relógios. Esse processo nunca será instantâneo, portanto no exato momento que "copiamos" o valor de um relógio para o outro, eles já estão defasados.

Se ao menos esse atraso fosse constante, era só atualizar já somando o atraso conhecido, mas não é.

Questão 5

Um computador pergunta para outro o horário. Ao receber o computador de referencia envia seu próprio horário e algumas informações a mais para ajudar a estimar o atraso. Ao receber o tempo, o computador que perguntou utiliza alguma técnica para estimar o atraso da mensagem. Uma idéia é supor que o atraso de ida é igual ao atraso de volta.

Questão 6

O NTP funciona colocando uma rede de computadores, com algum deles conectados diretamente ao UTC. O cliente que quer sincronizar pergunta a vários servidores e estima o atraso para cada um deles. Faz a média e obtem uma referência. Um ponto importante é que ele não atualiza o relógio diretamente, o que ele faz é modificar a taxa de ajuste do relógio (fazendo a hora passar mais rápido ou mais devagar), fazendo com que o relógio nunca volte no tempo e também que as mudanças de hora sejam suaves.

Questão 7

1. a e r.
2. h e m.
3. $b||k, b||n, b \rightarrow u, k||n, k||u, n||u$

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
1	2	3	4	6	10	11	1	3	4	5	8	1	4	8	9	12	2	3	5	7

4.

Evento	P1	P2	P3	P4
a	1	0	0	0
b	2	1	0	0
c	3	1	0	0
d	4	1	0	0
e	5	5	1	2
f	6	5	4	4
g	7	5	4	4
h	0	1	0	0
i	0	3	1	0
j	1	4	1	2
k	1	5	1	2
l	4	6	1	5
m	0	0	1	0
n	0	3	2	0
o	4	3	4	4
p	4	3	5	4
q	7	5	6	4
r	1	0	0	1
s	1	0	0	2
t	4	1	0	3
u	4	1	0	5

5.

6. Não existe. O Relógio de Vetor não desclassifica uma comparação do Relógio de Lamport. Isto é, $L(x) > L(y) \rightarrow V(x) > L(y)$. (Repare que o contrário não é verdadeiro)

7. Que $x||y$ ou $x \rightarrow y$

8. Que $x \rightarrow y$

Questão 8

TODO

Questão 9

Vantagem: É fácil provar e garantir a corretude, já que o coordenador controla e libera apenas 1 *Grant* de cada vez. **Desvantagem**: Ponto unico de falha. Sem o coordenador o sistema não funciona.

Questão 10

A vantagem é que em caso de *crash-failure* os nós vizinhos ao que falhou poderam se conectar e assim não fragmentará o anel, dando uma certa tolerância a falhas ao sistema todo. Conhecer mais do que dois nós aumentaria o nível de tolerância, aumentando o número de vizinhos que devem falhar ao mesmo tempo. Tudo depende das demandas de tolerancia a falhas.

Questão 11

São um conjunto de propriedades que são muito desejadas em sistemas transacionais para garantir alguma corretude do estado dos dados. * **A**: Atomicidade - Ou uma transação é totalmente efetivada ou totalmente cancelada. * **C**: Consistência - As transações devem seguir e manter algumas regras (definidas por sistema) sobre o estado global. * **I**: Isolamento - Uma transação não interfere na outra. No ponto de vista delas é como se elas estivessem executando sozinhas, mesmo que existam outras em paralelo. No final o resultado tem de ser de forma que as transações pudessem ter acontecido de forma serial. * **D**: Durabilidade - A partir do momento em que a transação se confirma o banco garante que os dados serão preservados.

Questão 12

Pode acontecer *deadlock* caso uma transferência de A para B aconteça em paralelo com uma de B para um C qualquer. Para que isso aconteça podemos adiquirir todos os locks no começo e usar alguma forma de ordenação entre as contas (pelo identificador delas, por exemplo).

Exemplo:

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

Questão 13

Two Phase Locking é uma técnica para evitar deadlocks e garantir atomicidade.

Usa-se dois tipos de locks: *write locks* e *read locks* para cada objeto do sistema.

- Um *read lock* pode ser adquirido ao menos que um *write lock* exista no momento para o mesmo objeto.
- O *write lock* pode ser obtido somente se nenhum outro lock existir no momento. (Isso permite várias leituras ao mesmo tempo).

Além disso os locks são adiquiridos e liberados em duas etapas:

- *Expanding Phase*: Todos os locks devem ser adiquiridos nessa etapa, que acontece logo no começo do procedimento.
- *Shrinking Phase*: Fase na qual os locks são liberados.
- Na *Strong Strict Two Phase Locking* todos os locks são liberados no final apenas.
- Na *Strict Two Phase Locking* os *read locks* podem ser liberados a qualquer momento.

Essas duas etapas são semelhantes ao que fizemos no exemplo acima.

Questão 14

Two Phase Commit é uma técnica para realizar transações com exclusão mútua e atomicidade em sistemas com dados distribuidos em vários nós.

Em cada transação, haverá um coordenador que ira gerenciar se a transação deve ou não ser executada.

Assim como em *2PL* também existem duas fases (talvez a unica semelhança):

- Preparação:
- Coordenador informa as sub-transações para cada participante.
- Cada participante adiquire os locks (2PL) para a sub-transação.
- Informa ao coordenador se tudo está certo para realizar a transação.
- Execução:
- Se o coordenador recebe ao menos um negação de transação, ele manda *abort* para todos os participantes.
- Se ele receber positivo de todos, ele manda *commit* para todos
- Ao receber *commit* os participantes executam de fato a transação, liberando os locks ao final.
- Caso os participantes recebam *abort*, eles apenas liberam os locks.

Questão 15

Não. Ele simplesmente existe para tentar garantir atomicidade e exclusão mútua.

Podemos fazer com que ao realizar uma transação os pedidos de votos sejam sempre feitos na mesma ordem (ordem dos participantes, talvez por PID ou outro identificador) Além disso podemos usar uma ordenação dos objetos e fazer 2PL sempre usando a mesma ordem (ordem dos objetos, talvez por ID do objeto na tabela, por exemplo)

O problema é que isso custaria muita performance (agora a fase de preparação seria sequencial e não mais paralela).

Questão 16

1. O coordenador não vai receber voto e vai mandar *abort* por timeout.
2. Quando o participante retomar a atividade ele pode perguntar a algum outro participante o que aconteceu e fazer o mesmo.
3. Ninguem faz nada, esperam o coordenador voltar.

Questão 17

Normalmente uma propriedade para provar isolamento é que o resultado das transações tenha alguma representação serial (mesmo que elas acontecam em paralelo). Sem os devidos cuidados alguns problemas podem acontecer:

- **Read-Write**: Quando uma transação lê duas vezes um mesmo objeto e obtem valores diferentes, o que nunca aconteceria numa execução serial (já que teoricamente somente um processo estaria executando ao mesmo tempo)

T1	T2
R(A)	
	R(A)
	W(A)
R(A)	

  T1 deveria ser abortado. *Strict Two Phase Locking*(S2PL) resolveria esse problema, não deixando T2 adiquirir o lock de escrita em A

- **Write-Read**: Quando dois processos acontecem "entrelaçados", persistindo parte do resultado de um e parte do resultado de outro. Algo que nunca aconteceria em uma execução serial.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

Ao final teriamos o A de T2 e o B de T1, o que não poderia acontecer. S2PL também resolve isso não deixando T2 obter os locks nem de A nem de B.

Esse caso é semelhante ao da transação da questão 12. Note que S2PL pode causar deadlocks, entretanto se tivermos uma ordenação dos objetos pode ser realizado sempre na ordem, assim como fizemos na questão 12.

- **Write-Write**: É a anomalia causada por processos que escrevem de forma alternada.

T1	T2
W(A)	
	W(B)
	W(A)
W(B)	

Isso seria um problema pois usando S2PL pela ordem de escrita causaria um deadlock. Nem sempre é possivel evitar deadlocks nesses casos.

Questão 18

TODO

Questão 19

- **Confiabilidade**: é o tempo que o sistema fica operacional até ter uma falha.
- **Disponibilidade**: é a fração de tempo que o sistema fica operacional.

Por exemplo, digamos que um sistema fique na média 9 dias funcionando até ter uma falha (MTTF = 9 dias) e que ele demore 1 dia para se recuperar da falha (MTTR = 1 dia).

Dado isso a *confiabilidade* é de 9 dias e a *disponibilidade* é de 90%.

Questão 20

Seja D_i a disponibilidade do sistema usando i componentes redundantes. Temos que:

$D_1 = \frac{MTTF}{MTTF+MTTR}$

$D_i = 1 - (1 - D_1)^i$

Logo

$D_i = 1 - (1 - \frac{MTTF}{MTTF+MTTR})^i$

Isso porque a probabilidade de um componente estar em falha é o complemento da disponibilidade dele. Além disso a probabilidade do sistema estar em falha é a probabilidade de todos falharem, portanto a probabilidade de estar em falha com i componentes, considerando que as falhas são independentes, é $(1 - D_1)^i$.

Sendo assim, se queremos $D_i = 99.99\%$, tendo $MTTF = 2.5 \cdot 365 \cdot 24 = 21900$ horas e $MTTR = 32$ horas, teremos:

$0.9999 = 1 - (1 - \frac{21900}{21932})^i$

$(\frac{32}{21932})^i = 0.0001$

$i = \log_{\frac{32}{21932}} 0.0001 = 1.41047291569$

Portanto são necessários 2 componentes redundates.

Questão 21

1. Depende do número de colunas. Pode ser que o sistema inteiro falhe.
2. Somente uma informação será repassada para a próxima coluna e os próximos votadores não repassarão, fazendo com que o sistema inteiro falhe.

Questão 22

Crash failures são fáceis de descobrir que aconteceram (ex.: servidor não responde). Falhas bizantinas podem passar despercebidas e pior, causar inconsistência no sistema (por exemplo, inconsistência nos dados).

Questão 23

TODO