STD29006 – Sistemas Distribuídos 07/11/2017

Projeto Prático 2: Sistema distribuído de *logs*

Professor: Emerson Ribeiro de Mello http://docente.ifsc.edu.br/mello/std

1 Descrição do problema

Neste projeto será feito uso de relógio lógico vetorial para imprimir na ordem correta as mensagens trocadas entre os processos em um sistema distribuído. Na tela do processo de *Log* deveria ser impresso algo como:

```
#Timestamp           FROM  Message   LogicalClock
2017-07-11 09:40:00  P1    m1        [1,0,0]
```

O *timestamp* que aparece na tela seria obtido a partir do relógio físico da máquina onde o processo *Log* está em execução. Esse *timestamp* indica a hora exata que a linha foi impressa na tela, mas não possui qualquer relação com horário que a mensagem fora recebida pelo processo *Log*.

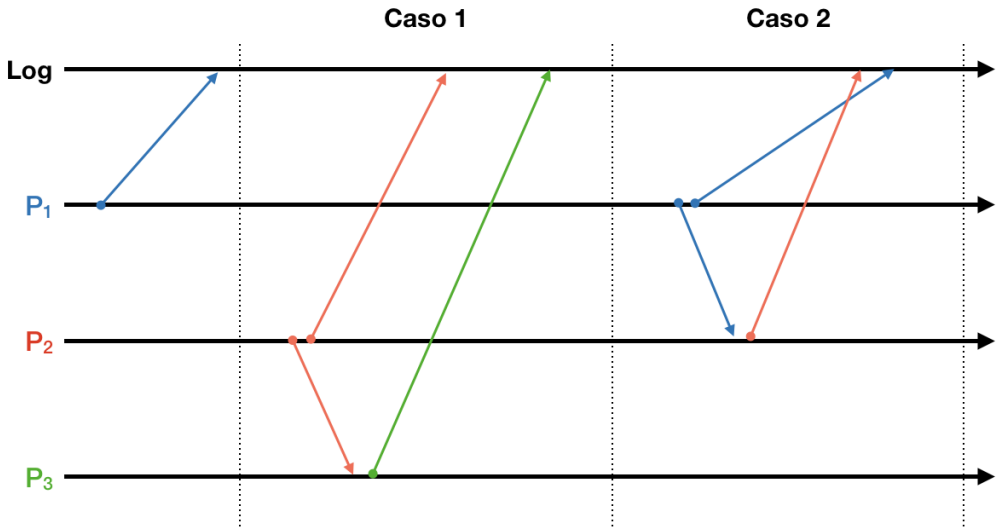

Figura 1: Exemplo de troca de mensagens entres os processos do sistema distribuído

Na Figura 1 é apresentado um exemplo de troca de mensagens entre os processos *workers* P_1, P_2, P_3 e o processo de registro *Log*. Quando um processo *worker* processa um evento (i.e. enviar ou receber mensagem), esse processo deve enviar uma mensagem para o processo *Log*. O processo *Log* consegue fazer uma ordenação FIFO das mensagens que são entregues a ele oriundas de um mesmo processo. Contudo, essa ordenação não seria suficiente em cenários onde a mensagem enviada por um processo P_i para *Log* é consequência de uma mensagem que P_i recebeu de P_j. Por exemplo, no caso 1 da Figura 1, o processo P_2 envia uma mensagem para P_3 e na sequência envia uma mensagem para *Log*. P_3, ao receber a mensagem de P_2, também envia uma mensagem para *Log*. Nesse exemplo, a ordem das mensagens entregues ao processo *Log* segue a mesma ordem de criação dessas mensagens. Contudo, no cenário 2 da Figura 1, a mensagem enviada por P_2 para *Log* chega antes da mensagem enviada por P_1 para *Log*, não respeitando assim a ordem dos eventos que geraram tais mensagens.

Este projeto prático terá duas entregas (versões v1 e v2). Faça uso das *tags*[1] do git para marcar os *commits* que representarão cada entrega. Esse projeto deverá ser desenvolvido com Java RMI.

[1] https://git-scm.com/book/pt-br/v1/Git-Essencial-Tagging

2 Versão v1 – sem uso de relógio lógico

2.1 Processo Log

Para essa versão o processo *Log* deverá estar apto a receber mensagens dos demais processos e deverá imprimir na tela as mensagens assim que forem recebidas. Nessa versão o processo *Log* deverá : (1) receber a lista de processos (*workers*) que poderão enviar mensagens para ele (essa lista deverá ser fornecida por meio de um arquivo texto, sendo que cada linha do arquivo estará o identificador único de um processo – *i.e.* p1); (2) receber os relógios lógicos junto com as mensagens. Apesar do processo *Log* receber o relógico lógico, esse ainda não fará uso do mesmo. Isso será explorado na versão v2.

2.2 Processo Worker

A lógica de um processo *worker* é: aguarde um tempo e processe um evento da lista de eventos fornecida por meio de um arquivo texto. O evento pode ser um evento local (**l**) ou pode ser o envio de uma mensagem (**e**) para um outro processo *worker*. Dessa forma, esse processo também deverá ser capaz de processar mensagens recebidas de outros processos. Para efeitos de depuração, use o método `System.err.println()` para imprimir cada evento processado.

Para cada evento (local, envio de mensagem ou recebimento de mensagem), o processo deve enviar uma mensagem para o processo *Log*. Para essa versão ainda não será necessário trabalhar com o relógio lógico. Sendo assim, o relógio lógico a ser enviado pode conter 0 em todas as posições.

Cada instância de processo worker deverá: (1) possuir um identificador único; (2) uma semente única para a geração dos números pseudo-aleatórios (um número inteiro); (3) informações para localização dos demais processos (Log e demais *workers*); (4) um arquivo com a lista de eventos que esse deverá processar; (5) um valor em milissegundos para o tempo que o processo ficará aguardando antes de processar o próximo evento; (6) um valor em milissegundos para variação de atraso (*jitter*) máximo entre o envio de uma mensagem para um processo *worker* e o envio de uma mensagem para o processo *Log*. O *jitter* para cada mensagem será obtido por meio de um número sorteado de 0 até o valor máximo fornecido. Todas essas informações deverão ser fornecidas como argumentos de linha de comando. Exemplo:

```
# java nomeDaClasse Id Semente tempoEspera tempoJitter arquivo-com-processos arquivo-com-
    eventos
java processoWorker p1 123456 3000 2000 processos.txt eventos.txt
```

O formato do arquivo `processos.txt` poderia ser:

```
p1
p2
p3
Log
```

No arquivo `eventos.txt` deve-se colocar cada evento em uma linha. Cada linha será formada pelo identificador do evento (**l** – para eventos locais e **e** – para mensagens que deverão ser enviadas) e pelo conteúdo do evento. O conteúdo do evento deverá ser um número inteiro (de 0 a 10000) obtido de forma aleatória Exemplo:

```
l,1
e,2
e,3
l,4
e,5
```

Cada mensagem é enviada para um processo destino escolhido de forma aleatória. Um processo *worker* só pode iniciar o processamento do arquivo de eventos quando todos os demais processos já estiverem sido instanciados. Sendo assim, deve-se fazer um sincronizador para garantir isso (um laço que não permitirá a progressão até que os demais estejam instanciados). Abaixo o pseudo-código para o processo *worker*.

```
sincronizarProcessos(); // só prosseguirá se todos os processos estiverem instanciados

while(...){
  aguardarTempoAleatorio();
  processarEvento(); // lido do arquivo ou recebido de outro processo
  jitter();
  enviarParaLog();
}
```

2.3 Primeiro experimento

Para validar as implementações realizadas nas Subseção 2.1 e Subseção 2.2 monte um cenário com 4 processos *workers* e execute alguns experimentos. Verifique se a ordem das mensagens impressas na tela do processo *Log* está de acordo com a ordem que essas mensagens foram criadas, lembrando que um processo *worker* ao enviar ou receber uma mensagem para/de um processo outro processo *worker*, esse também deverá enviar uma mensagem para o processo *Log*.

3 Versão v2 – fazendo uso de relógio lógico

Nessa versão o processo *worker* deve manter um relógio vetorial próprio e encaminhar esse relógio junto com qualquer mensagem que enviar para outro processo (*Log* ou *worker*). Para tal, deve-se implementar a seguinte interface:

```
public interface RelogioVetorial{
  public int[] atualizar(int idDoProcessoAtual, int[] relogio, int[] relogioRecebido);
}
```

Se o processo *Log* imprimir na tela toda mensagem assim que ele a receber, então esse poderá imprimir em uma ordem errada. Se o processo *Log* optar por receber todas as mensagens, guardá-las em uma fila e só imprimir na tela depois de todos os processos *workers* forem encerrados, então poderíamos ter aqui a ordem correta.

Se desejarmos imprimir as mensagens na ordem correta e com os processos *workers* ainda em execução, então poderíamos fazer uso de uma fila de espera de mensagens no processo *Log* e este ainda deveria possuir um relógio para registrar os *timestamps* das últimas mensagens recebidas de cada processo *worker*. Quando uma nova mensagem chega no processo *Log*, este deve atualizar o relógio, armazenar a mensagem na fila de espera e percorrer a fila em busca de mensagens que poderiam ser impressas.

3.1 Segundo experimento

Para validar as modificações realizadas realizadas na Seção 3 execute o mesmo cenário que foi montado na Subseção 2.3. Verifique se a ordem das mensagens impressas na tela do processo *Log* está de acordo com a ordem que essas mensagens foram criadas.

⚠ Atenção:

Data para entrega: ***28/11/2017****, via Github Classroom. O projeto poderá ser desenvolvido em grupos com até dois alunos.*

Referências

[1] Johan Montelius and Vladimir Vlassov, *Loggy: a logical time logger*, 2016.