Resolução do Exercício dos *Raids*

J. Pereira P. Almeida

21 de janeiro de 2024

1 Enunciado

Considere um sistema para organizar *raids* num jogo multi-jogador. Cada jogador indica qual o número mínimo de participantes no *raid*. Os participantes no próximo *raid* ficam definidos mal existem jogadores à espera em número suficiente para satisfazer os requisitos de todos eles (em termos de número de participantes). Os jogadores que apareçam mais tarde serão agrupados no *raid* seguinte. Logo que possível, o sistema 1) indica a cada jogador os nomes dos outros jogadores desse *raid*; 2) quando o *raid* pode começar pois o sistema só permite que estejam até R *raids* a decorrer em simultâneo.

Apresente duas classes Java (para serem usadas no servidor) que implementem as interfaces abaixo, tendo em conta que os seus métodos serão invocados num ambiente *multi-threaded*.

```java
public interface Manager {
    Raid join(String name, int minPlayers) throws InterruptedException;
}
```

```java
public interface Raid {
    List<String> players();
    void waitStart() throws InterruptedException;
    void leave();
}
```

A operação `join` deverá bloquear até estar formado o grupo de participantes no *raid*, devolvendo o objeto que o representa; tem como parâmetro o nome do jogador e o número mínimo de jogadores que o *raid* deve ter. A operação `players` devolve a lista de jogadores presentes no *raid*; `waitStart` deverá bloquear até o *raid* poder começar (só podem estar R a decorrer em simultâneo); `leave` é invocada quando um jogador abandona o *raid*, que termina quando todos os jogadores o tiverem feito.

Simplificação: Apresente apenas uma classe Java com a interface `Manager` em que o método `join` devolve diretamente a lista de participantes (`List<String>` em vez de `Raid`) e ignore o limite R. Deve no entanto cumprir os requisitos em termos de número de participantes.

2 Problema simplificado

Começamos por resolver a versão simplificada do enunciado em que devemos implementar apenas a seguinte interface:

```
5  public interface Manager {
6      List<String> join(String name, int minPlayers)
7              throws InterruptedException;
8  }
```

Como primeiro passo, temos que perceber quando é que cada fio de execução tem que esperar e qual a condição que permite que faça progresso. Neste problema, cada fio de execução ao invocar a operação `join` irá indicar qual o número mínimo de jogadores com os quais aceita participar num *raid*. Como exemplo, vamos admitir que chegam jogadores que indicam como mínimos aceitáveis, respetivamente, 3, 1 e 2. Como resultado:

- o primeiro jogador tem que ficar à espera, pois precisa de pelo menos 3 jogadores para iniciar;
- o segundo jogador só precisa de 1 para iniciar mas como o enunciado indica que só pode ser iniciado um *raid* se existirem jogadores em número suficiente para satisfazer os requisitos de todos os que estão à espera, só poderá iniciar quando estiver também satisfeito o minimo do primeiro, que ainda está à espera de 3;
- o terceiro jogador só precisa de 2 para iniciar e como já há dois à espera, em que nenhum deles precisa mais do que 3, deve agora ser iniciado o *raid*.

Desta análise tiramos duas conclusões:

- os jogadores juntam-se a um *raid* estritamente por ordem de chegada, o que simplifica o problema pois não temos que considerar estratégias complicadas para os agrupar;
- o *raid* pode ser iniciado logo que o número dos jogadores à espera seja igual ao máximo dos mínimos pretendidos.

Podem pois existir situações extremas. No caso do primeiro jogador a chegar para um *raid* indicar um mínimo de 1, ele poderá iniciar imediatamente um *raid* sozinho. Por outro lado, se um jogador indicar um número muito elevado, impedirá o início do *raid* durante muito tempo. Apesar destas situações poderem, intuitivamente, ser indesejáveis, nada no enunciado pede para que sejam evitadas e tratadas como casos especiais.

2.1 Primeira aproximação

Podemos assim escolher variáveis que nos permitam avaliar a condição pela qual cada fio de execução tem que esperar:

```
11     private int maxMin;
12     private List<String> players = new ArrayList<>();
```

Na variável `maxMin` guardamos o máximos dos mínimos pretendidos por cada um dos jogadores que está à espera, cujos nomes guardamos em `players`. Note-se que só precisamos da lista, em vez de um simples contador, porque a operação `join` precisa de devolver a lista de jogadores que constituem o *raid*.

Como vamos manipular estado partilhado e ter fios de execução que esperam por eventos, precisamos tambem de, pelo menos, um trinco e uma variável de condição:

```
8     private Lock l = new ReentrantLock();
9     private Condition cond = l.newCondition();
```

Podemos então propôr uma primeira implementação da operação `join`:

```
14    public List<String> join(String name, int minPlayers)
15            throws InterruptedException {
16        l.lock();  ①
17        try {
18            players.add(name); ②
19            maxMin = Math.max(maxMin, minPlayers);
20            if (players.size() == maxMin) {
21                cond.signalAll(); ③
22            } else {
23                while (players.size() < maxMin) ④
24                    cond.await();
25            }
26            return players;
27        } finally {
28            l.unlock();  ⑤
29        }
30    }
```

que efetua os seguintes passos:

- começamos por fechar o trinco ① porque vamos manipular estado partilhado;
- atualizamos então o estado partilhado adicionando o nome a `players` e calculando o novo máximo dos mínimos pretendidos em `maxMin` ②;
- como modificamos variáveis que estão a ser usadas numa condição de espera, deveriamos acordar fios de execução na variável de condição correspondente, mas só vale a pena fazê-lo depois de verificar que de facto vale a pena ③;
- caso contrário, temos que esperar ④;
- finalmente, abrimos o trinco para que a secção crítica possa ser usada por outros.

Esta primeira aproximação não resolve totalmente a versão simplificada do problema, uma vez que consegue formar apenas um primeiro *raid*. Retomando o mesmo exemplo, se chegar um quarto jogador que pretenda participar num *raid* de até 4 jogadores, ele será acrescentado aos jogadores do primeiro *raid* e não esperará por mais jogadores como seria desejado.

Se tentarmos resolver isto colocando `maxMin = 0` e uma lista vazia em `players` logo que o *raid* fica completo ③, estamos a alterar os valores de que depende a condição ④ que será re-testada pelos jogadores do primeiro *raid*, depois de acordarem. Por acaso, a condição permanecerá falsa ($0 \not< 0$) e parece que a solução funciona. Mas esse é o caso apenas se o `join` tiver tempo de completar para todos os jogadores do primeiro *raid*. Quando um jogador para o próximo *raid* invocar `join` com um `minPlayers` > 1 (o mais normal) a condição passará a verdadeira. Se tal acontecer enquanto existem ainda jogadores do primeiro *raid* a acordar, quando estes forem testar a condição ④, esta já será de novo verdadeira, pelo que continuarão em espera. Este erro pode não ser detetado durante muito tempo.

2.2 Segunda aproximação

Poderiamos acrescentar variáveis para identificar e guardar separadamente o estado de diferentes *raids* (e.g., ter um inteiro sequencial que identifica o *raid* e mapas com a informação para cada um deles), de forma a que cada jogador saiba qual o *raid* em que vai participar e verifica a condição sobre o estado correspondente. Além da complexidade, isto faria com que tivessemos que resolver um problema adicional, que é eliminar o estado relativo a *raids* dos quais já nenhum jogador está à espera.

Podemos obter uma segunda aproximação que já resolve corretamente o problem observando que, como o Java é uma linguagem orientada por objetos, deve ser possível criar uma instância da classe da secção anterior, de forma a que o estado de diferentes *raids* seja separado.

```
7  public class ManagerImpl implements Manager {
8      private Lock l = new ReentrantLock();
9      private Condition cond = l.newCondition();
10
11     private RaidImpl current = new RaidImpl(); ⑥
12
13     public List<String> join(String name, int minPlayers)
14             throws InterruptedException {
15         l.lock();
16         try {
17             RaidImpl raid = current;
18             raid.players.add(name);
19             raid.maxMin = Math.max(raid.maxMin, minPlayers);
20             if (raid.players.size() == raid.maxMin) {
21                 current = new RaidImpl(); ⑦
22                 cond.signalAll();
23             } else {
24                 while (raid.players.size() < raid.maxMin) ⑧
25                     cond.await();
26             }
27             return raid.players;
28          } finally {
29             l.unlock();
30         }
31     }
32
```

```
33      private class RaidImpl { (9)
34          List<String> players = new ArrayList<>();
35          int maxMin = 0;
36      }
37  }
```

As alterações em relação à solução inicial aproximada são:

- guardamos o estado relevante para cada *raid* numa classe aninhada `RaidImpl` (9) e a instância atual, do *raid* que está a ser organizado, numa variável `current` (6);
- sempre que um *raid* fica completo, criamos uma nova instância (7) para ser usada por todos os jogadores que chegarem depois;
- mesmo depois de completar um *raid*, as variáveis referentes ao anterior continuam válidas, no objeto respetivo, e serão usadas pelos fios de execução que acordam para avaliar a condição (8). Para tal é importante a referência ao *raid* a ser organizado quando o jogador invocou `join`, guardado em `raid`, e não ao `current`, que pode entretanto ter mudado para o próximo *raid*.

2.3 Solução otimizada

Na solução anterior podemos observar que a condição (8) pode ser reescrita como `raid == current`, o que faz com que os fios de execução que estão a dormir deixem de precisar do valor de `maxMin` para conseguir avaliar a condição.

Sendo assim, não precisamos de guardar `maxMin` separadamente para cada *raid* e, neste caso, não precisamos da classe `RaidImpl` para guardar a única variável que sobra, `players`. Efetuando estas alterações obtemos a seguinte solução compacta:

```
7   public class ManagerImpl implements Manager {
8       private Lock l = new ReentrantLock();
9       private Condition cond = l.newCondition();
10
11      private int maxMin;
12      private List<String> currentPlayers = new ArrayList<>();
13
14      public List<String> join(String name, int minPlayers)
15              throws InterruptedException {
16          l.lock();
17          try {
18              List<String> players = currentPlayers;
19              players.add(name);
20              maxMin = Math.max(maxMin, minPlayers);
21              if (players.size() == maxMin) {
22                  currentPlayers = new ArrayList<>();  (10)
23                  maxMin = 0;
24                  cond.signalAll();
25              } else {
26                  while (players == currentPlayers)  (11)
27                      cond.await();
```

```
28              }
29              return players;
30          } finally {
31              l.unlock();
32          }
33      }
34  }
```

Concretamente, esta versão difere das anteriores na medida em que:

- quando um *raid* fica completo, reinicializamos a informação criando uma nova lista para armazenar os jogadores do *raid* seguinte, assinalando assim que o anterior está completo (10) e podemos então mudar `maxMin = 0` porque esta variável não volta a ser consultada pelos fios de execução que vamos agora acordar;
- alteramos a condição (11) para comparar a lista de jogadores do *raid* em formação com a lista em que inscrevemos este jogador que sendo diferentes indicam que o *raid* correspondente já está completo.

Dica

Esta abordagem, usando a comparação da referência para o objeto na condição de espera, é genérica e pode ser usada sempre que nos deparemos com este problema de ter uma solução para uma única iteração de um problema que não é facilmente reiniciada mas que precisamos de usar repetidamente.

3 Versão completa

A resolução do enunciado completo traz dois problemas adicionais. Em primeiro lugar o método `join` precisa de devolver uma implementação de uma segunda interface que dá acesso a funcionalidade adicional, na qual se inclui a obtenção da mesma lista de jogadores que já tinhamos. De facto, considerando a solução da Secção 2.1, isto será facilmente atingido se fizermos a classe aninhada auxiliar implementar a interface `Raid`.

O segundo desafio corresponde a uma nova necessidade de sincronização de forma a evitar que haja mais do que `R` *raids* em curso. Esta funcionalidade é no entanto bastante comum em programação concorrente e está descrita em detalhe no Capítulo 3 dos apontamentos, correspondendo a um semáforo. De forma a perceber exatamente em que situações temos que fazer fios de execução esperar, analisamos com mais cuidado o enunciado:

- Sem qualquer dúvida, cada jogador terá que começar por usar o método `join` para obter uma instância de `Raid`.
- Tendo obtido um *raid*, poderá em princípio usar qualquer um dos métodos. Vamos no entanto assumir que o método `leave` não será usado por um fio de execução antes de ter invocado o `waitStart` correspondente. Isto não é claro no enunciado, pelo que somos livres de escolher esta interpretação que, como veremos mais tarde, nos simplifica a resolução.

- Não é também claro do enunciado qual deve ser o resultado do método `players` depois de algum dos jogadores desse *raid* já ter invocado `leave`. Mais uma vez vamos escolher a opção que nos simplifica a implementação, assumindo que este método devolve sempre a lista dos jogadores iniciais do *raid*.

3.1 Primeira aproximação

A primeira aproximação à solução completa passa pois por usar uma classe aninhada como na Secção 2.1:

```
35      private class RaidImpl implements Raid { ①
36          List<String> players = new ArrayList<>();
37          boolean started = false;
38          int playing; ②
39
40          void init() {
41              players = Collections.unmodifiableList(players); ③
42              playing = players.size(); ④
43          }
44
45          public List<String> players() {
46              return players; ⑤
47          }
48
49          public void waitStart() throws InterruptedException {
50              /*  ... */
51          }
52
53          public void leave() {
54              l.lock();
55              playing -= 1; ⑥
56              l.unlock();
57          }
```

- Esta classe aninhada agora não serve simplesmente para guardar estado auxiliar mais vai implementar a interface `Raid` ① que tem que ser devolvida pelo método `join`.
- Como será necessário saber quando o *raid* começa e quantos jogadores ainda não deixaram o *raid*, para saber quando ele termina e com isso, quantos *raids* ainda estão em curso, acrescentamos variáveis ② que inicializamos no momento em que o *raid* fica formado ④. Note-se que não podemos usar aqui o construtor, pois o objeto `RaidImpl` é criado quando o *raid* está ainda sem jogadores e começa a ser formado.
- Podemos então decrementar o número de jogadores ainda ativos sempre que um deles anunciar que sai ⑥, caso em que precisamos de fechar o trinco pois pode ser executado por fios concorrentes. Sublinha-se aqui a possibilidade que o Java dá de usar variáveis da classe exterior a partir da classe aninhada, neste caso, `l`.

É interessante ainda referir que no método `players` não precisamos de fechar o trinco para obter a lista de jogadores ⑤. Isto acontece porque a partir do momento em que o *raid* está formado e que este método pode começar a ser usado por diferentes fios de execução, a lista `players` nunca mais é modificada. Além disso, a ultima modificação à variável ocorreu protegida por um trinco `l` que precisa de ser visitado por outros fios de execução para obterem o objeto `RaidImpl` antes de usarem o método `players`, pelo que está garantida também a visibilidade das modificações. É no entanto necessário garantir que a lista permanece imutável, impedindo que um jogador a possa modificar depois de devolvida. Para isso guardamos no objeto uma cópia da lista original que seja garantidamente imutável ③.

3.2 Segunda aproximação

Podemos agora resolver o resto do problema garantindo que não há mais do que `R` *raids* ativos. Para o conseguir, precisamos de saber quantos *raids* estão em curso e, de forma a garantir que eles começam por ordem, uma fila de espera para os que não puderam começar por já haver `R` ativos:

```
11      public static final int R = 10;
12      private int running = 0;
13      private Queue<RaidImpl> pending = new ArrayDeque<>();
```

Podemos assim, no momento em que um *raid* fica completo, verificar se o podemos iniciar imediatamente ou se temos que o colocar em espera:

```
40      void tryStart(RaidImpl raid) {
41          if (running < R) {
42              running += 1;
43              raid.start();
44          } else {
45              pending.add(raid);
46          }
47      }
```

Quando um *raid* termina, se houver algum em espera, tentamos iniciá-lo:

```
49      void finished() {
50          running -= 1;
51          RaidImpl raid = pending.poll();
52          if (raid != null)
53              tryStart(raid);
54      }
```

Note-se que não precisamos nestes métodos fechar explicitamente o trinco apesar de estarmos a manipular estado partilhada, uma vez que eles não são públicos e podem apenas ser invocados a partir de outros métodos da mesma classe (ou da classe nela aninhada) que já o fizeram.

Podemos agora completar os métodos que faltam, que correspondem ao caso mais simples da espera por um evento num fio de execução distinto que é assinalado por uma variável booleana:

```
70          void start() {
71              started = true; ⑦
72              cond.signalAll();
```

```
73          }
74
75          public void waitStart() throws InterruptedException {
76              l.lock();
77              try {
78                  while (!started)  ⑧
79                      cond.await();
80              } finally {
81                  l.unlock();
82              }
83          }
84
85          public void leave() {
86              l.lock();
87              playing -= 1;
88              if (playing == 0)
89                  finished(); ⑨
90              l.unlock();
91          }
```

- O início de um *raid*, que é decidido ao nível da class `ManagerImpl`, traduz-se na modificação de uma variável e no acordar de todos os fios de execução que estejam à espera. Uma vez que usamos apenas uma única variável de condição para o gestor e para todos os *raids*, nesta situação iremos acordar potencialmente muitos fios de execução que não conseguirão fazer progresso, o que não é ótimo. O enunciado não pedia no entanto explicitamente que este problema fosse evitado.
- Cada jogador pode assim esperar por este início observando a variável `started` e dormindo na variável de condição.
- Quando o último jogador sai do *raid*, assinala-se o seu fim para que o gestor possa iniciar algum outro que esteja à espera. Note-se que se tivessemos optado que o método `players` devolvesse a lista atualizada de jogadores, tinhamos aqui que atualizar essa lista descobrindo em primeiro lugar qual era o jogador que estava a sair.

3.3 Solução otimizada

Uma vez que o código da secção anterior já foi estruturado de forma a que as variáveis do gestor e de cada *raid* sejam manipuladas em métodos dessa mesma classe, torna-se fácil otimizar o código usando vários trincos e variáveis de condição de forma a minimizar a contenção em trincos e a possibilidade de fios de execução serem acordados em variáveis de condição quando não vão fazer progresso. Obtemos assim uma solução completa e otimizada:

```
7   public class ManagerImpl implements Manager {
8       private Lock l = new ReentrantLock();
9       private Condition cond = l.newCondition();
10
11      public static final int R = 10;
12      private int running = 0;
13      private Queue<RaidImpl> pending = new ArrayDeque<>();
```

```
14
15      private RaidImpl current = new RaidImpl();
16      private int maxMin = 0;
17
18      public Raid join(String name, int minPlayers) throws InterruptedException {
19          l.lock();
20          try {
21              RaidImpl raid = current;
22              raid.players.add(name);
23              maxMin = Math.max(maxMin, minPlayers);
24              if (raid.players.size() == maxMin) {
25                  raid.init();
26                  tryStart(raid);
27                  maxMin = 0;
28                  current = new RaidImpl();
29                  cond.signalAll();
30              } else {
31                  while (current == raid)
32                      cond.await();
33              }
34              return raid;
35          } finally {
36              l.unlock();
37          }
38      }
39
40      void tryStart(RaidImpl raid) {
41          if (running < R) {
42              running += 1;
43              raid.start();
44          } else {
45              pending.add(raid);
46          }
47      }
48
49      void finished() {
50          l.lock(); (10)
51          running -= 1;
52          RaidImpl raid = pending.poll();
53          if (raid != null)
54              tryStart(raid);
55          l.unlock();
56      }
57
58      private class RaidImpl implements Raid {
59          Lock rl = new ReentrantLock();
60          Condition rcond = rl.newCondition();
61
62          List<String> players = new ArrayList<>();
63          boolean started = false;
64          int playing;
65
66          void init() {
67              players = Collections.unmodifiableList(players);
```

```
68              playing = players.size();
69          }
70
71          public List<String> players() {
72              return players;
73          }
74
75          void start() {
76              rl.lock();  (11)
77              started = true;
78              rcond.signalAll();
79              rl.unlock();
80          }
81
82          public void waitStart() throws InterruptedException {
83              rl.lock();
84              try {
85                  while (!started)
86                      rcond.await();
87              } finally {
88                  rl.unlock();
89              }
90          }
91
92          public void leave() {
93              rl.lock();
94              playing -= 1;
95              if (playing == 0)
96                  finished();
97              rl.unlock();
98          }
99      }
```

Precisamos agora de ter cuidado quando invocamos métodos entre o gestor e os *raids*, concretamente:

- Quando iniciamos um *raid* (11) temos que obter o trinco `rl`, pois temos apenas o trinco `l` correspondente ao gestor e pode já haver outros fios de execução a invocar o método `waitStart`.
- Quando finalizamos um *raid* (10) temos que obter o trinco `l` do gestor, pois temos apenas o trinco `rl` do *raid* que acabou de terminar e o gestor pode estar a ser usado por fios de execução de outros raids, que têm outras instâncias de `rl`.

Uma vez que estamos a fechar simultaneamente dois trincos e não seguimos sempre a mesma ordem, temos que estar atentos à possibilidade de impasses. Neste caso, isso poderia acontecer se existir um fio de execução que corresponde ao último jogador e está a invocar o método `finished` ao mesmo tempo que o método `start` do mesmo *raid* está a ser invocado pelo gestor. No entanto, como assumimos que um jogador só usa o `leave` depois de `waitStart` e sabemos que não pode sair de `waitStart` antes de `start` ter sido invocado, então temos a garantia que não temos impasses. Se isto não fosse verdade, deviamos

impedir que os dois trincos fossem obtidos por ordens diferentes, por exemplo, fazendo com que `leave` abrisse `rl` antes de invocar `finished`.

Uma solução intermédia de menor complexidade, evitando estes potenciais problemas, mas mantendo o objetivo de evitar acordar fios de execução sem necessidade, seria ter apenas um trinco, no gestor, e uma variável de condição por *raid*, todas associadas a esse trinco. Mas tal levaria a mais contenção. Como está, `waitStart` não causa contenção no gestor, e `leave` apenas quando o *raid* termina (`playing` chega a 0), e não em cada invocação.