Cloud Computing

Trabalho Prático - Sistemas Distribuídos

Universidade do Minho, Departamento de Informática
Rodrigo Monteiro, Diogo Abreu, Filipa Pinto, e Flávio Silva
{a100706, a100646, a96862, a97352}@alunos.uminho.pt

1. Introdução

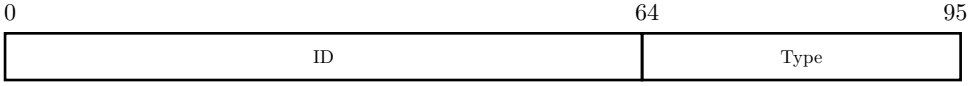
Neste projeto, implementamos um serviço de *cloud computing* com funcionalidade *Function-as-a-Service:* clientes enviam código de tarefas de computação para um servidor principal, que faz uma distribuição para outros servidores, especializados na execução de tarefas, de acordo com a memória necessária para execução, e de acordo com as configurações de memória desses servidores – a que chamamos *workers*.

Para isso, utilizamos a linguagem Java, *threads* e *sockets* TCP, respeitando os seguintes requisitos: uma única conexão entre cada duas máquinas envolvidas; um protocolo de comunicação em formato binário; e cada *thread* do servidor associada a apenas um *socket*. E tendo como objetivos minimizar o número *threads* acordadas, diminuir a contenção e assegurar que não ocorre *starvation*.

Implementamos, também, interfaces simples para o cliente e para os *workers* com o padrão *Model-View-Controller*.

2. Protocolo

Todas as mensagens protocolares definidas começam com dois campos: um identificador, ID, e o tipo de mensagem. Sendo o ID do tipo primitivo `long`, e o tipo da mensagem identificado através de um `int`.



Assim, todas as classes que representam as mensagens protocolares são subclasses de uma classe `Packet`, que possui os atributos `id`, e `PacketType` – uma interface que o tipo de mensagem tem de implementar.

```
public abstract class Packet {
    private final long id;
    private final PacketType type;
    // ...
public interface PacketType {
    int getValue();
    static void serialize(PacketType type, DataOutputStream out) throws IOException
    { out.writeInt(type.getValue()); }
    static <T extends Enum<T> & PacketType> T
    (DataInputStream in, Class<T> enumType) throws IOException, IllegalArgumentException
    { /* ... */ }
}
```

Decidimos agrupar os métodos de serialização e deserialização em classes próprias. Caso os métodos fossem implementados nas subclasses de `Packet`, estes seriam úteis se fossem `static`, o que não resultaria com herança, e levaria a repetição de código.

```
public class ClientPacketDeserializer implements Deserializer {
    public Packet deserialize(DataInputStream in) throws IOException { ... }
}
```

2.1. **Client packets**

Os clientes enviam pacotes do tipo *Registration, Login, Job,* e *Status*. Podem fazer um registo, fornecendo um nome único e uma palavra-passe, ou podem efetuar login. Depois disso, podem enviar pedidos de execução de tarefas, verificar pedidos enviados ou recebidos, ou fazer logout.

- Registration e Login
  - *Name:* Nome único do cliente (tamanho variável)
  - *Password:* Palavra-passe relativa ao registo (tamanho variável)
- Logout
- Job request
  - *Required memory:* Memória necessária para executar o código da tarefa (8 bytes)
  - *Data length:* Comprimento do array de bytes (8 bytes)
  - *Data:* Array de bytes que representa o código da tarefa (tamanho variável)
- Status request

2.2. **Server packets**

O servidor principal é responsável por receber e gerir os pedidos de registo, login e logout, tal como gerir uma fila de tarefas, e a distribuição destas através das conexões com os *workers*.

- Information (enviado para os clientes)
  - *Memory limit:* Limite máximo de memória que uma tarefa pode ter (8 bytes)
  - *Total memory:* Memória total, isto é, a soma da memória dos servidores conectados (8 bytes)
  - *Used memory:* Memória a ser utilizada pelos servidores (8 bytes)
  - *Queue size:* Tamanho da fila de tarefas (4 bytes)
  - *Nº connections:* Número de clientes conectados (4 bytes)
  - *Nº workers* Número de servidores conectados (4 bytes)
  - *Nº workers waiting:* Número de servidores à espera de tarefas (4 bytes)
- Job request (enviado para os *workers*)
  - *Client name:* Nome do cliente que enviou o pedido de execução da tarefa (tamanho variável)
  - *Required memory:* Memória necessária para executar o código (8 bytes)
  - *Data length:* Comprimento do array de bytes (8 bytes)
  - *Data:* Array de bytes que representa o código da tarefa (tamanho variável)
- Job result (enviado para os clientes)
  - *Result status:* Indica se foi possível executar a tarefa (4 bytes)
  - *Error message:* Caso não tenha sido possível executar a tarefa, é enviada a mensagem de erro produzida (tamanho variável)

  - *Data length:* Tamanho do output produzido (8 bytes)
  - *Data:* Array de bytes do output (tamanho variável)
- Status (enviado para os clientes)
  - *Status:* Identificador do estado de um pedido do cliente (4 bytes)

2.3. Worker packets

Um *worker* envia inicialmente um pedido de *Connection*, e é responsável por receber pedidos de execução de tarefas, executar as tarefas e enviar os resultados para o servidor principal. Por fim, envia um pedido de *Disconnection*, para informar o servidor principal que deixa de estar disponível.

- Connection
  - *Memory:* Memória disponível do servidor para a execução de tarefas. (8 bytes)
  - *Nº threads:* Número de worker threads. (4 bytes)
- Disconnection
- Job Result
  - *Client name:* Nome do cliente que pediu a execução da tarefa (tamanho variável)
  - *Result status:* Identifica se foi possível executar a tarefa (4 bytes)
  - *Error message:* Caso não tenha sido possível executar a tarefa, é enviada a mensagem de erro produzida (tamanho variável)
  - *Data length:* Tamanho do output produzido (8 bytes)
  - *Data:* Array de bytes do output (tamanho variável)

3. Implementação

3.1. Client

A classe `Client` implementa a seguinte interface, *ClientAPI*:

```
public interface ClientAPI {
    void createRegistration(String name, String password);
    long sendRegistration() throws IOException;
    long sendLogin() throws IOException;
    long sendLogout() throws IOException;
    long sendJob(int requiredMemory, byte[] job) throws IOException;
    long sendGetInfo() throws IOException;
    Packet receive(long id) throws IOException, InterruptedException;
    Packet fastReceive(long id) throws IOException, InterruptedException;
    List<Packet> getJobRequests();
    List<Packet> getJobResults();
    void exit() throws IOException;
}
```

Para além disso, possui os atributos: `ClientPacketSerializer` que utiliza para serializar e enviar mensagens para o servidor; `ServerPacketDeserializer` que utiliza para deserializar as mensagens que recebe do servidor; `Demultiplixer` que utiliza para receber mensagens do servidor, organizando-as por ID, permitindo que se espere por uma ou mais mensagens com um ID específico; `JobManager` que utiliza para ler a diretoria com o código das tarefas, e para guardar os resultados recebidos em ficheiros numa dada diretoria; entre outros.

Na implementação, não foi necessário receber mais do que uma mensagem com o mesmo ID. Apesar disso, optamos por utilizar uma `ConditionQueue<Packet>` para cada ID como uma medida proativa para garantir flexibilidade.

3.2. **Server**

3.2.1. Conexões

O servidor principal possui dois tipos de conexões, conexões com clientes e conexões com *workers*.

3.2.2. Gestão das tarefas

3.2.2.1. Worker server: Measure Selector Queue

Nesta versão, utilizamos uma `queue` personalizada a que chamaos `MeasureSelectorQueue`: uma lista duplamente ligada, que adiciona elementos no fim e retira do início, tendo em conta uma determinada condição (se o elemento não verificar a condição a lista é percorrida sequencialmente do primeiro ao último elemento até encontrar um elemento correspondente), e mantém uma `min-heap`.

Uma classe chamada `SharedState` gere essa fila, e as instâncias de `WorkerConnection` fazem `dequeue`dessa fila, passando como argumento o seu limite de memória. As `WorkerConnection` adquirem uma tarefa quando têm memória suficiente, e quando conseguem obter uma `lock`.

```
// Na classe WorkerConnection
Job job = this.sharedState.dequeueJob(this.maxMemory);
// ...
while (job.getRequiredMemory() + this.memoryUsed > this.maxMemory)
  this.hasMemory.await();

// Na classe SharedState
public Job dequeueJob(long maxMemory) {
  try {
      this.ljobs.lock();

      while (jobs.isEmpty(maxMemory))
          this.hasJobs.await();

      Job job = this.jobs.poll(maxMemory);
      this.notFull.signal();
      // ...
  } finally { this.ljobs.unlock(); }
}
```

3.2.2.2. Worker server: Ordered Queue

3.3. **Worker**

4. Funcionamento

5. Conclusões e trabalho futuro

References

1. Coulouris, G., Dollimore, J., Kindberg, T., Blair, G.: Distributed Systems: Concepts and Design. Pearson Education Limited, Edinburgh Gate, Harlow, Essex, CM20 2JE, England (2012)
2. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2012)