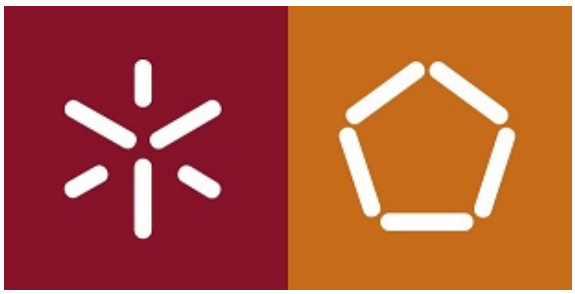


Universidade do Minho
Departamento de Informática

AirGroup11

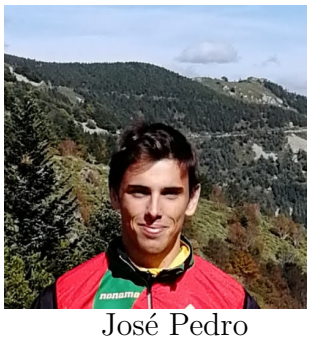
Sistemas Distribuídos

Grupo 11

15 de janeiro, 2022


Catarina
(a93319)


Henrique Parola
(a93325)


José Pedro
(a93325)


Vasco
(a93206)

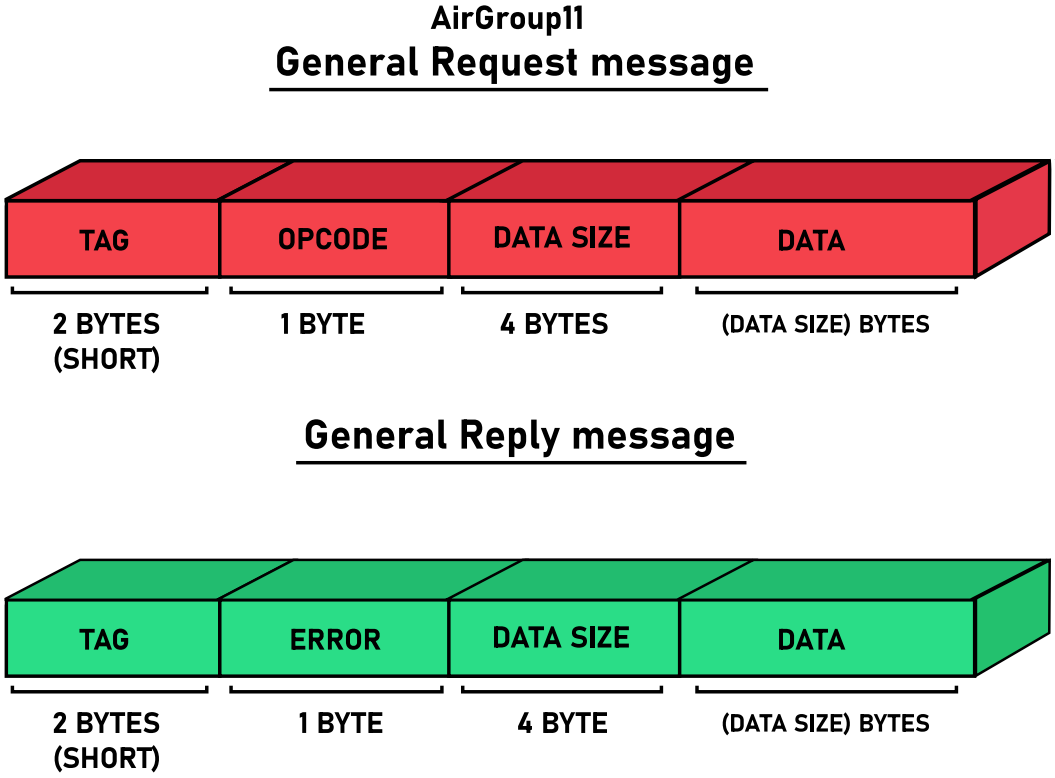
Índice

1. Mensagens protocolares

1.1 Aspetos gerais

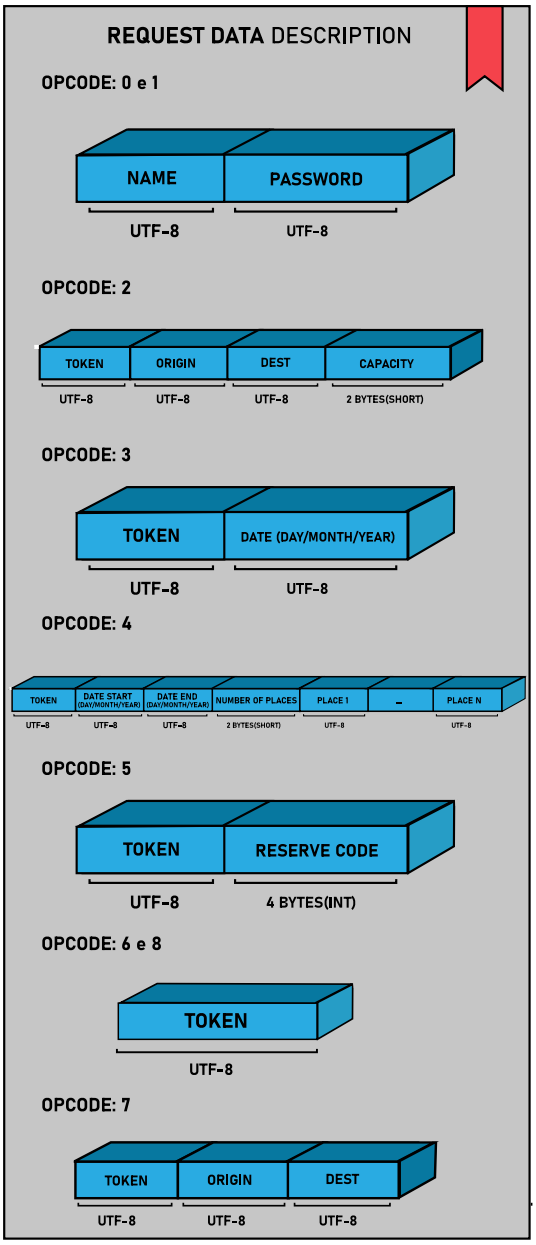
De modo a garatir uma linguagem única para o **cliente** e o **servidor** se comunicarem, a primeira etapa do trabalho consistiu na construção de mensagens protocolares. Já com as mensagens definidas, foi possível a implementação de forma independente da aplicação cliente e do servidor.

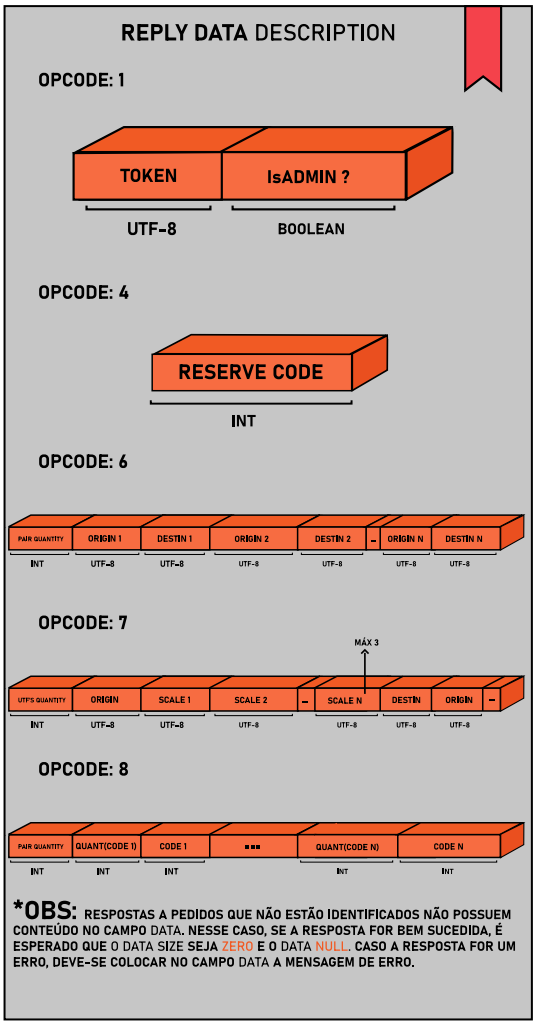
Tendo em conta a possibilidade de haver clientes *multithreading*, o *design* mais geral das mensagens protocolares deviam ter um campo para a identificação do pedido e da resposta a enviar ao cliente - tal campo foi intitulado **TAG**. Seguidamente definiu-se um *byte* para o **OPCODE** da mensagem, isto é, o seu significado (no caso de ser um *Request*) ou um *byte* para o controlador de erros vindo do servidor (no caso de ser uma *Reply*). A escolha de um *byte* para o **OPCODE** justifica-se por haver no máximo 9 tipos de mensagens protocolares. Dito essas características, segue então uma visualização gráfica das mensagens de *Request* e *Reply* criadas:



1.2 Especificação das mensagens

Restava então a descrição do conteúdo (campo **data**) de cada tipo de mensagem. Segue então a especificação dos conteúdos, tanto para *Request* e *Reply*.



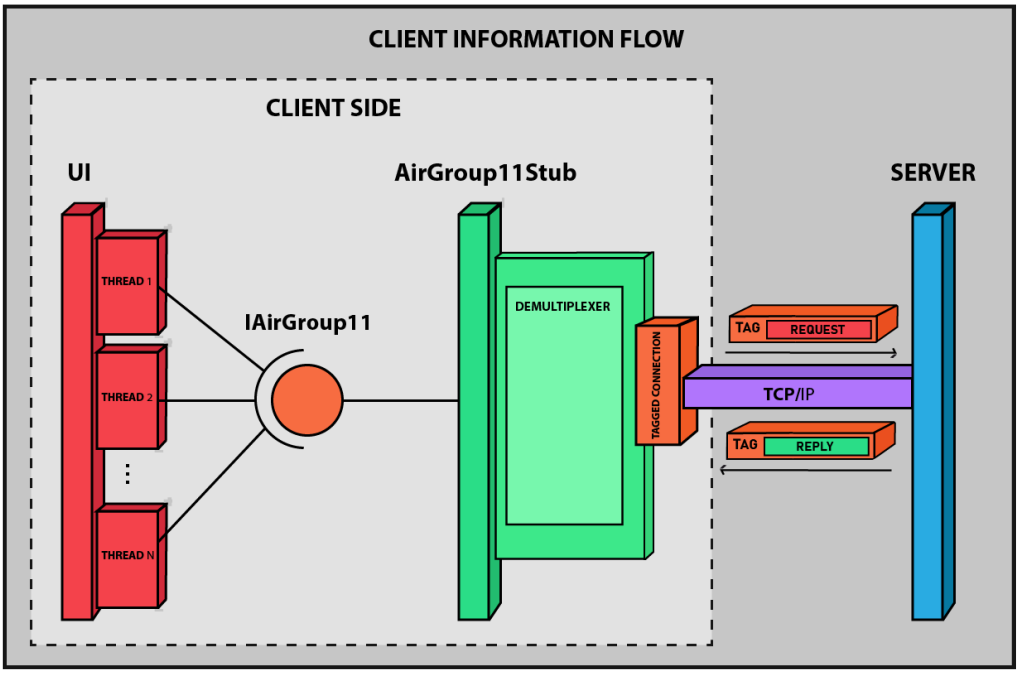


A definição das mensagens mostradas, cobre todos os requisitos propostos no enunciado do trabalho, como também adiciona uma nova funcionalidade, que é a possibilidade de poder pedir ao servidor todos os códigos de reserva efetuadas por parte de um cliente.

2. Cliente

2.1 Arquitetura

A arquitetura em camadas utilizada para a aplicação cliente possibilitou o isolamento do código que usa um serviço que reside num sistema remoto. Por outras palavras, construímos uma abstração similar ao JAX RPC. Com isso foi possível desenvolver a *User Interface* sem ter em conta se o servidor é local ou remoto. Isto porque a **UI** utiliza um objeto que implementa a interface **IAirGroup11**, sem saber se o objeto em questão implementa um serviço remoto. Com isso conseguimos desacoplar código distribuído de não distribuído e garantir uma boa independência entre as camadas.



No nosso caso, a classe **AirGroup11Stub** implementa serviços remotos. Logo, a **UI** invoca métodos em um *stub* que faz conexão com o servidor. Para além disto, a aplicação cliente desenvolvida é *multithreading*. Logo, o serviço remoto desenvolvido, para fornecer suporte a *multithreading*, utiliza um demultiplexador para ser capaz de identificar e distribuir os pedidos e respostas de cada *Thread*.

Acreditamos ter então alcançado um dos objetivos de um sistema distribuído que é a **transparência de acesso**, sendo indistiguível para a **UI** se o serviço utilizado é local ou remoto.

2.2 Demultiplexador

O Demultiplexador implementado disponibiliza o método:

```
1 public Reply service(Request request);
```

A ideia de tal método é podermos fazer um pedido e aguardar a resposta do servidor. Tendo em conta que o *stub* é um objeto que pode estar sendo utilizado em mais de uma *Thread*, foi tomado o cuidado de garantir o controle da concorrência para a utilização do **service()**. Outro ponto importante a referir é que optamos por deixar para o demultiplexador a responsabilidade de atribuir *tag's* aos pedidos, de forma automática e única para cada pedido. Com isso possibilitamos que uma mesma *Thread* possa voltar a utilizar o demultiplexador e também elimina a preocupação de criação de *tag's* estáticas aos pedidos. Segue então um excerto do método **service()**, aquando do envio do pedido:

```
1 l.lock();
2 tagSend = currentTag;
3 Condition c = this.l.newCondition();
4 conditions.put(tagSend,c);
5 tagged.send(tagSend,request);
6 currentTag++;
```

A criação de uma condição para cada pedido facilita o processo de "acordar"a *Thread* correspondente à resposta que chegar do servidor. Isto porque o demultiplexador possui uma *Thread* que está continuamente a receber respostas do servidor, colocando-as num *buffer* e acordando as *Threads* de acordo com a *tag* da resposta chegada. Segue um excerto de como é feito o recebimento das respostas:

```
1 Reply reply = tagged.receive();
2 if (reply != null){
3     l.lock();
4         try {
5             replies.put(reply.getTag(), reply);
6             conditions.get(reply.getTag()).signal();
7 // ...
```

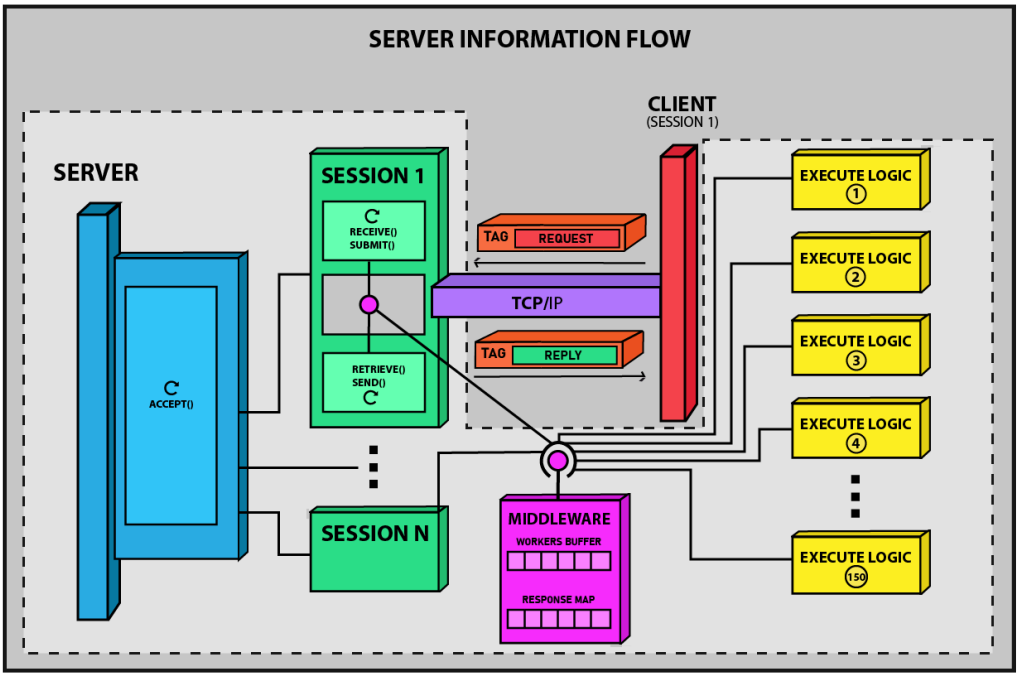
Por fim, voltando ao método **service()**, após ser feito o envio do pedido mostrado anteriormente, a *Thread* em questão fica em pausa até chegar a sua resposta.

```
1 while (reply == null){
2     try {
3         conditions.get(tagSend).await();
4         reply = replies.get(tagSend);
5 // ...
```

3. Servidor

3.1 Arquitetura

A implementação do servidor foi dividida em 3 camadas: **Sessão**, **Middleware** e **Lógica**. Quando um cliente deseja-se se conectar inicialmente ao servidor, é criada uma sessão para ele. A partir de então, todos os seus pedidos são direcionados para esta sessão. A camada da sessão é responsável então por receber os pedidos de cada utilizador e colocá-los num estado compartilhado - o **Middleware** - pelos 150 *workers* do servidor. Tais *workers* foram nomeados como **ExecuteLogic**. Para além disto, a sessão também é encarregue de enviar as respostas aos clientes, após os *workers* do servidor finalizarem os tratamentos dos pedidos.



Como pode ser observador, controlamos o processamento de dados com 150 *workers*, garantindo assim um limite para processamento em simultâneo. Outro detalhe é o armazenamento dos pedidos numa *queue*, garantindo justiça na ordem de tratamento dos pedidos dos clientes.

A autenticação ocorre no *Middleware* através de *Tokens* que são enviados aos clientes aquando do *login*. Como o *token* faz parte do *payload* dos *Requests*, apenas usuários que tiverem feito *login* são capazes de interagir com a sessão, previnindo assim ataques de terceiros.

3.2 Middleware

O Middleware é fundamental num sistema distribuído. Através de chamadas a uma interface minimalista é possível abstrair a complexidade necessária para certas operações alheias ao processamento lógico de um sistema distribuído. Esta foi a nossa filosofia ao implementar um middleware que possibilita funcionalidades como um lockmanager, autenticação por tokens e armazenamento temporário de pedidos e respostas. A interface `IMiddleWare` disponibiliza os seguintes métodos:

```
1 public String putToken(String username, String pwd);
2 public String getUserName(String token);
3 public void submit(SerializerFrame f, int sessionID);
4 public void putResponse(ReplySerializerFrame reply, int
     sessionID);
5 public ReplySerializerFrame retrieve(int sessionID);
6 public Map.Entry<Integer, SerializerFrame> bufferConsume();
7 public LockManager newLockManager();
```

O método `putToken` permite adicionar um novo token quando se faz login. A informação guardada é uma entrada num mapa com chave token e valor username. O método `getUserName` permite obter informação do username associado a um determinado username. Algo bastante importante, por exemplo, para muitos métodos da lógica de negócio. O método `submit` insere numa fila de pedidos (um BoundedBuffer genérico) a informação do id da sessão que faz invocação do método e o respetivo request. O método `bufferConsume` permite a recolha desta última informação inserida. Ação importante, por exemplo, para os workers quando se encontram disponíveis para iniciar o processamento de um novo pedido. O método `putResponse`, permite uma forma de retorno da resposta por parte dos workers. É através da camada do middleware que os workers conseguem devolver o resultado da seu processamento de volta à sessão. O método é invocado inserindo a reply e o id da sessão do pedido processado, num mapa cuja chave é o id da sessão e o valor um BoundedBuffer genérico que guarda as replies. Desta forma, na sessão é possível obter o resultado através da invocação do método `retrieve`.

3.3 LockManager

O LockManager tem como principal intuíto a gestão dos locks associados a objetos. É uma solução que permite apenas guardar o estado de locks associados a determinados objetos. Um estado de lock é algo que identifica quantas chamadas ativas ao lock foram feitas e tem informação relativa a cada ação de lock feita. Ao fazer lock de um objeto que não tem um lockstate associado é criado um novo lockstate e adicionado à lista de lock ativos um certo estado. Este estado contém o tipo de modo que se pretende executar o lock e o id da thread que executou o lock. O modo pode ser `Mode.S` (Shared) ou então `Mode.X` (Exclusive). Quando se fizer unlock, através do identificador da thread é possível retirar o estado correto da lista de locks ativos e decrementar o número de "chamadas ativas". Se se verificar que este número passa a nulo, então pode-se remover o estado da memória para não sobrecarregar o sistema. Desta forma é possível ter um nível de granularidade superior sem o overhead de ter um lock para todos os objetos mesmo para aqueles que não estão em uso.

4. Comentários finais e conclusão

A nível de funcionalidades para a aplicação, foram implementados todos os requisitos propostos, assim como a adição do mecanismo de poder consultar todos os códigos de reservas efetuadas. Para além disto, acreditamos ter conseguido conceber uma excelente arquitetura para o sistema, tanto para o servidor como para o cliente, permitindo suporte a clientes *multithreading* e um servidor não vulnerável a clientes lentos. A arquitetura em camadas desenvolvida também concretiza uma boa separação de código distribuído para não distribuído (no cliente) e código que respeita a lógica de negócio separado de operações alheias a tal (no servidor).

Colocamos o servidor à prova com um *script* de teste que cria 281 aplicações clientes em paralelo. Cada cliente cria um registo diferente de utilizador, seguidamente faz *login* e tenta reservar um mesmo voo, cuja capacidade é 140 (dando margem de 3 dias de intervalo). O resultado foi o esperado, houve a alocação de 140 reservas para o voo num dia, 140 noutro e houve apenas uma reserva para o terceiro dia proposto.

No que respeita a trabalho futuro, acreditamos também no possível melhoramento de algumas partes do sistema. Por parte do cliente, poderá ser feito uma melhor gestão de processamento, atribuindo um limite de trabalhos em paralelo. Para além disto, uma separação da serialização/deserialização dos diversos tipos de *Requests* e *Replies* poderá ser remodelada de forma a criar classes apenas para realizar tais tarefas, aliviando assim código diretamente no *stub*. Tais sugestões mencionadas não foram implementadas (nesta entrega) por parte da aplicação cliente, dado não haver problemas de escalabilidade dada a dimensão do sistema cliente a ser construído. Em contrapartida, por parte do servidor, o cuidado tomado foi redobrado, sendo dada mais atenção para a sua gestão de processamento.

Por parte do servidor poderia ser feito um LockManager que através de uma árvore de dependências consegui-se fazer um lock granular. Tal estratégia foi tentada, mas devido a resultados não favoráreis e à falta de tempo não foi possível fazer o debug necessário para por a ferramenta a funcionar. Podia-se ainda ter feito um mecanismo de notificações, algo que com a nossa arquitetura não seria muito difícil de implementar.