A blue and black logo

Description automatically generated

Instituto Politécnico de Setúbal

Escola Superior de Tecnologia de Setúbal

José Almeida (202100661)

Tiago Silva (202000331)

Licenciatura em Engenharia Informática

Computação Paralela e Distribuída

Contents

[Arquitetura do Sistema 3](#_Toc168246276)

[Implementação 3](#_Toc168246277)

[Pylint 4](#_Toc168246278)

[Extras adicionados 5](#_Toc168246279)

# Arquitetura do Sistema

O sistema é composto pelos seguintes componentes:

* **server**: Implementa o *server* JSON-RPC.
* **client**: Implementa o *client* JSON-RPC.
* **functions**: Contém as funções que o *server* disponibiliza.
* **tests\_server**: Contém testes unitários para validar o funcionamento do *server*.
* **tests\_client**: Contém testes unitários para validar o funcionamento do *client*.

# Implementação

**Servidor (server.py)**

* **Inicialização**: O servidor é inicializado criando uma instância da classe JSONRPCServer, que é configurada com o endereço e a porta de rede onde irá escutar as conexões do *client*.
* **Registro de Funções**: As funções que o servidor disponibiliza são registradas usando o método register(name, function). Exemplos de funções registradas incluem hello, greet, add, sub, mul, e div.
* **Execução do Servidor**: O método start() inicia o servidor, que entra em um loop onde aceita conexões do *client*. Cada conexão é tratada pelo método handle\_client(conn), que lê as mensagens recebidas, processa os pedidos JSON-RPC, invoca as funções registradas com os parâmetros fornecidos e envia as respostas de volta ao *client*.
* **Tratamento de Mensagens**: No método handle\_client(conn), as mensagens recebidas são decodificadas do formato JSON, validadas e processadas. Se o método solicitado estiver registrado, ele é invocado com os parâmetros fornecidos e o resultado é enviado de volta ao *client*. Caso contrário, uma mensagem de erro é retornada.

**Cliente (client.py)**

* **Inicialização**: O *client* é inicializado criando uma instância da classe JSONRPCClient, que é configurada com o endereço e a porta do *server* ao qual se conecta.
* **Envio de Solicitações**: O *client* envia solicitações para o *server* utilizando o método invoke(method, params). Este método cria uma mensagem JSON-RPC com um identificador de pedido, o método a ser invocado e os parâmetros. A mensagem é enviada ao *server* e a resposta é processada.
* **Recebimento de Respostas**: O cliente espera a resposta do *server*, que pode ser um resultado de sucesso ou uma mensagem de erro. A resposta é decodificada do formato JSON e o resultado é retornado ao chamador. Se houver um erro, uma exceção apropriada é levantada.

# Pylint

Uma imagem com texto, captura de ecrã, Tipo de letra, Paralelo

Descrição gerada automaticamenteUma das principais recomendações do Pylint foi substituir a classe genérica de exceção Exception por classes mais específicas. A recomendação foi aplicada na função invoke presente no arquivo client.py.

Foram também adicionadas as exceções customizadas:

Uma imagem com texto, captura de ecrã, Tipo de letra, número

Descrição gerada automaticamente

Tendo em conta a complexidade cognitiva, observamos que a função original handle\_client apresentava uma complexidade cognitiva de 22 devido à presença de múltiplas ramificações e exceções aninhadas. A função foi dividida em várias funções menores para reduzir a complexidade cognitiva e melhorar a legibilidade do código. As novas funções são responsáveis por processar o pedido, executar o método, e retornar os erros específicos.

A função handle\_client original continha também uma variável req que não era utilizada após sua atribuição. A variável req foi substituída pela variável \_, indicando que seu valor não é utilizado.

# Extras adicionados

**Parâmetros com nome:**

Uma imagem com texto, captura de ecrã, número, Tipo de letra

Descrição gerada automaticamente

Para implementar o suporte a parâmetros nomeados, fizemos as seguintes alterações:

* **Modificações no *server*** (server.py):
  + O *server* foi ajustado para diferenciar entre parâmetros posicionais e nomeados. Se os parâmetros forem fornecidos como um dicionário (dict), o *server* os passa como parâmetros nomeados para a função correspondente. Caso contrário, ele os passa como parâmetros posicionais.
  + Isso foi feito na função execute\_method, que agora verifica o tipo de parametros e chama a função adequada.
* **Modificações no *client*** (client.py):
  + O *client* foi ajustado para permitir a invocação de métodos com parâmetros nomeados. A função \_\_getattr\_\_ agora suporta tanto argumentos posicionais quanto nomeados e os envia corretamente no formato JSON-RPC.
  + Isso permite que o utilizador do *client*especifique diretamente.

**Várias funções num único pedido:**Uma imagem com texto, eletrónica, captura de ecrã, software

Descrição gerada automaticamente

Para implementar o suporte de várias funções num único pedido (Batch Requests), fizemos as seguintes alterações:

* **Modificações no** ***server*** (server.py):
  + O *server* foi ajustado para identificar se a solicitação é uma lista de chamadas (Batch Requests) ou uma única chamada. Em caso de Batch Requests, ele processa cada chamada individualmente e retorna uma lista de respostas.
* **Modificações no *client*** (client.py):
  + O *client* foi ajustado para permitir a criação e envio de Batch Requests. O método invoke\_batch foi adicionado para este propósito.