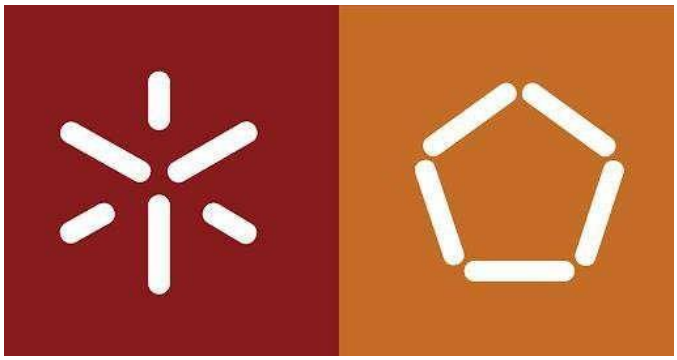

Universidade do Minho

Licenciatura em Engenharia Informática

Ano Letivo 2024/25

Comunicações por Computador

Trabalho Prático nº2

Realizado por: Gonçalo Cunha A104003, Gonçalo Cruz A104346, Nuno Ribeiro A104177

21/11/2024

Arquitetura da solução

A arquitetura da nossa solução garante organização e eficiência na execução das tarefas descritas no ficheiro JSON. A solução é composta por dois componentes principais: um servidor central, responsável por carregar o ficheiro JSON, interpretar as tarefas nele presentes e distribuí-las para os agentes; e os agentes, que executam as tarefas e reportam métricas de desempenho e estado, bem como verificam se certas condições específicas de desempenho são excedidas, enviando alertas (através do protocolo TCP) quando isso acontece. A comunicação principal entre o servidor e os agentes é realizada usando o protocolo UDP, recorrendo a *sockets*. Esta arquitetura foi desenvolvida de forma modular, o que resulta numa maior organização e permite uma fácil expansão e manutenção, e aproveita bibliotecas de Python para manipulação de dados e gestão da comunicação, como *json* e *threading*.

Especificação dos protocolos propostos
a. formato das mensagens protocolares

Para a comunicação servidor-agente foram desenvolvidos dois protocolos: NetTask, que usa UDP, para a comunicação de tarefas e para a coleta contínua de métricas; e AlertFlow, utilizando TCP, para a notificação de alterações críticas no estado dos dispositivos de rede. Para cada um destes protocolos, tivemos também de implementar cabeçalhos que refletissem as necessidades de representação e envio de informações (como ID, número de sequência, dados, etc.).

No módulo Message_Sending temos diversas funções responsáveis pelo envio de pacotes. Estas usam cabeçalhos que foram criados especificamente para cada tipo de envio, tendo em conta as informações necessárias para este fim.

Para enviar um **pedido de registo**, são necessários 8 bits para o ID do agente que está a enviar esse pedido, 3 bits para o tipo da mensagem (no caso é 0), os 16 bits seguintes são para o número de sequência e os últimos 5 bits foram acrescentados de modo a que o total de bits do cabeçalho seja um múltiplo de 8.

Para o servidor enviar um **ACK para um agente**, o cabeçalho é igual ao anterior, em que a diferença é o *msg_type* ser igual a 1. No entanto, caso o ACK seja para a **confirmação de receção de um *output*,** o cabeçalho já é diferente, tendo 8 bits para indicar o ID da tarefa, 3 bits para o *msg_type* (que é 4, neste caso) e 21 bits para o ID do *output* (quando o agente recebe este cabeçalho, o número de sequência é calculado como sendo um número com 7 dígitos em que os 2 dígitos mais significativos representam o ID da tarefa e os restantes o ID do *output*). Já para o envio de um **ACK para o servidor**, o cabeçalho tem novamente o formato indicado para o envio de um ACK para um agente.

Relativamente ao envio de uma **tarefa para um agente**, temos 8 bits para o ID do agente recetor (que não está efetivamente a ser utilizado, funcionando, neste caso, como *padding*), 3 bits para o tipo da mensagem (que é 2), 16 bits para o número de sequência (que é o ID da tarefa somado de 1), 3 bits para o tipo da métrica a ser coletada, 10 bits para a frequência com que o agente deve enviar dados de *output* e, por fim, são adicionados bits conforme o tipo de métrica da tarefa. Caso seja uma tarefa para obter as percentagens de CPU ou de RAM utilizadas não são adicionados mais bits ao cabeçalho. No entanto, caso seja para obtenção de dados de latência são adicionados 8 bits para indicação do número de pacotes, 8 bits para a frequência com que os pacotes são enviados e 32 bits para indicação do IP de destino do *ping* (é importante realçar que os dados de latência são obtidos infinitamente, com o comando *ping* a ser executado em intervalos de tempo correspondentes à frequência). Caso a métrica seja *bandwidth*, *jitter* ou *packet_loss* apenas são adicionados 64 bits para que sejam indicados os IPs do servidor e cliente *iperf*. Quanto à obtenção do estado de uma dada interface, são adicionados 8 bits para a representação da mesma. Para além disto, caso a métrica a obter não seja latência ou largura de

banda são ainda adicionados 8 bits para indicar o valor a partir do qual é enviado um alerta (sobre variações no desempenho).

Quanto ao envio de um **output por parte de um agente** são utilizados 8 bits para o ID de origem, 3 bits novamente para o *msg_type* (3, aqui), 24 bits para o número de sequência, 8 bits para o tipo da métrica, 32 bits para representar o valor obtido na última medição e caso a métrica não seja latência nem estado das interfaces são acrescentados 32 bits para a média das métricas obtidas.

b. diagrama de sequência da troca de mensagens

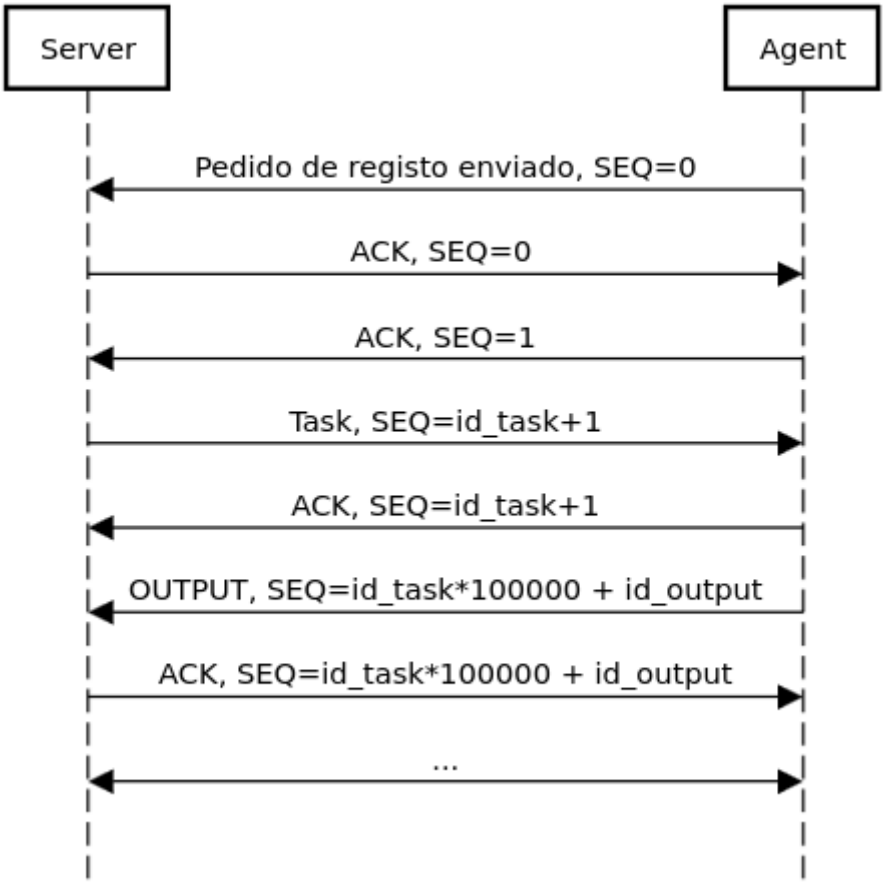

Figura 1 - Diagrama de sequência de troca de pacotes entre servidor e agente

Nesta figura apresenta-se um diagrama representativo das trocas de mensagens entre servidor e agente. Pode-se ver que a comunicação começa com um pedido de registo, enviado pelo agente. Quando o servidor o recebe, envia uma confirmação (ACK) ao agente, com número de sequência 0. Por sua vez, este responde com um novo ACK, desta vez com número de sequência 1. De seguida, o servidor envia uma tarefa para esta ser executada pelo agente com o qual está a comunicar, que, aquando da receção da mesma, envia um novo ACK de confirmação. Tem-se ainda o envio do output da tarefa, novamente com o envio de uma confirmação do lado do recetor (neste caso o servidor). Pode-se ter ainda outros pacotes, de tarefas ou outputs e respetivos ACKs, tendo ainda em conta que também podem ocorrer perdas, que são lidadas do lado do emissor, até à receção do ACK correspondente. Quanto a situações de Alert_Flow, os pacotes indicativos dos erros são enviados por parte do agente, novamente com confirmação por parte do servidor.

Implementação

a. detalhes, parâmetros, bibliotecas de funções, etc.

Como referido anteriormente, tivemos o cuidado de separar o código em diferentes módulos: Menu, Message_Sending, NMS_Server, NMS_Agent, Output, Parser.

Para o **Menu**, temos apenas uma função, menu(), que trata de imprimir as opções que o utilizador da aplicação pode escolher (entre qual agente deseja ver os outputs e qual métrica deseja visualizar), bem como obter a opção inserida pelo mesmo e chamar a função exibe_output (presente no módulo Output).

No **Message_Sending** temos as funções referidas no capítulo anterior.

Quanto ao **NMS_Server**, temos funções para processamento de mensagens e criação do servidor em UDP e TCP. A função processamento_udp recebe como parâmetros *message,* que é a mensagem recebida do agente; *agent_address*, que é, como o nome indica, o endereço IP do agente; *s*, que é o *socket* do agente, responsável pela comunicação; e *tarefas_sem_ack*, que é uma lista onde são armazenados os números de sequência das mensagens enviadas para cada agente. A esta lista é adicionado um número de sequência assim que uma tarefa é enviada e é removido assim que o servidor recebe o ACK correspondente, sendo que também é usada para a receção de pedidos de registo, em que o número de sequência é adicionado à lista assim que envia o ACK de confirmação, e é removido assim que receber o ACK do agente, e para receção de outputs, onde, mais uma vez, é adicionado à lista quando recebe o output mas, neste caso, não chega a removê-lo, uma vez que esse número será utilizado para o servidor não coletar o mesmo output repetidamente. Esta lista é usada, em alguns casos, para o *server* repetir o envio dos pacotes (intercalados por um *timeout*) até receber uma confirmação vinda do agente. Esta função é responsável por gerir a comunicação UDP com agentes, descodificando as mensagens recebidas, enviando tarefas ou ACKs, recolhendo métricas e registando outputs nos respetivos ficheiros. Recorre também a *threads* para enviar mensagens e processar tarefas em paralelo.

Na função udp_server os parâmetros são *address*, que é o IP do router onde o servidor é inicializado, e *port*, que é a porta do mesmo router. É responsável pela inicialização de um servidor via UDP e cria uma *thread* para lidar com cada mensagem recebida. Para a parte do TCP, temos novamente uma função processamento_tcp, que tem como argumentos um *socket* para receção de mensagens vindas de um agente e agent_adress, que é o IP do respetivo agente. Esta função imprime no terminal todas as notificações de alerta que recebe. A tcp_server recebe um *address* do server e a respetiva porta (*port*). Esta inicia um server em TCP.

Relativamente ao **NMS_Agent**, temos uma função auxiliar run_command que, como o nome indica, executa o comando recebido como argumento (*command*), a métrica a ser analisada, o valor de alert_flow para o qual é enviada notificação de alerta, o *socket*, o endereço IP, a porta e o ID do agente, o ID da tarefa a executar, a lista de mensagens onde os números de sequência das

mensagens trocadas são guardados para fins de retransmissão (de funcionamento semelhante à lista do servidor) e tcp_errors, que é a lista onde as notificações de alerta são guardadas. Esta função monitoriza e processa a saída de um comando em tempo real, extraindo métricas, verificando condições de alerta e enviando resultados ao servidor. Também esta cria uma *thread* por cada mensagem enviada. Temos ainda a udp_agent, que recebe (para além dos argumentos comuns ao udp_server) o identificador do agente e a lista que vai ser passada à função anteriormente referida. É responsável por implementar um agente UDP que se regista no servidor, recebe tarefas para execução e responde com *outputs* ou confirmações. A comunicação é garantida com recurso a *timeouts*, *threads* para operações paralelas e mecanismos de retransmissão de mensagens perdidas. Temos também a função tcp_agent, novamente semelhante à tcp_server, que recebe ainda a lista tcp_errors, onde, para cada erro nela presente, envia-o ao servidor. Por fim, este módulo possui a função auxiliar notificacao_tcp, que serve para definir a mensagem de alerta que será enviada para o servidor, em que, dependendo do tipo de métrica (metrica), essa mensagem pode ou não incluir o valor da ultima_metrica (valor que desencadeou o envio da mensagem). Tem ainda como parâmetros o ID do agente que obteve esse valor e o valor limite (alert_flow), usado pela maior parte das métricas.

No módulo **Output** temos a função criar_ficheiro_output, que tem como parâmetros o identificador do router em que o agente está a executar, o tipo de métrica, a última métrica obtida, a média das métricas calculadas até ao momento (que apenas não é usado na medição de latência e estado das interfaces) e ainda o identificador relativo ao *output*. Esta função serve para criar o ficheiro (em JSON), caso ainda não exista, onde o output vai ser escrito ou então fazer *append* caso este já exista. O nome do ficheiro de output é “outputs<id>” onde <id> é substituído pelo identificador do router, recebido pela função. A limpar_ficheiro_output, que serve apenas para eliminar o conteúdo dos ficheiros de output, sendo que esta função é chamada sempre que o servidor é inicializado. Temos também a exibe_output, que recebe o identificador do agente, para poder aceder ao ficheiro de outputs do mesmo, e o tipo de métrica, que indica a métrica que se pretende imprimir.

Quanto ao módulo **Parser**, a função ler_ficheiro_json abre o ficheiro recebido como argumento (caminho_ficheiro) e guarda na variável dados o conteúdo do mesmo (caso ele exista), retornando esses mesmos dados. A função parser_task extrai os parâmetros contidos na tarefa task_from_server e gera um comando para ser executado, que é retornado juntamente com a métrica e o valor de alert_flow (exceto para latência e largura de banda). O argumento agent_addresses contém todos os endereços IP associados ao router em que o agente está instalado e servem para a função saber se, no comando *iperf*, o agente executará como cliente ou servidor. Já a parse_output extrai de uma linha de output os dados necessários para envio ao servidor, recorrendo à variável metrica. Recebe ainda metricas_totais, que é uma lista que armazena todos os valores já obtidos para a métrica em questão. A função parse_alert recebe qual o tipo de métrica que originou o disparo de alerta, o respetivo valor obtido e o id da tarefa onde essa métrica era avaliada e é responsável por criar um cabeçalho (com 8 bits reservados para o id da tarefa,

8 bits para o tipo de metrica e 32 bits para o valor correspondente) que será enviado pelo agente para o servidor.

Por fim, relativamente a bibliotecas utilizadas temos socket, json, threading, subprocess, entre outras. A primeira foi utilizada para a criação dos canais de comunicação de servidor e agentes, a json serviu para manipulação e extração de dados dos ficheiros JSON, para criação de *threads* recorremos à threading e a subprocess serviu para criar os processos que executam, na prática, os comandos fornecidos.

Testes e resultados

De seguida apresentam-se exemplos de tarefas das várias métricas que são possíveis obter.

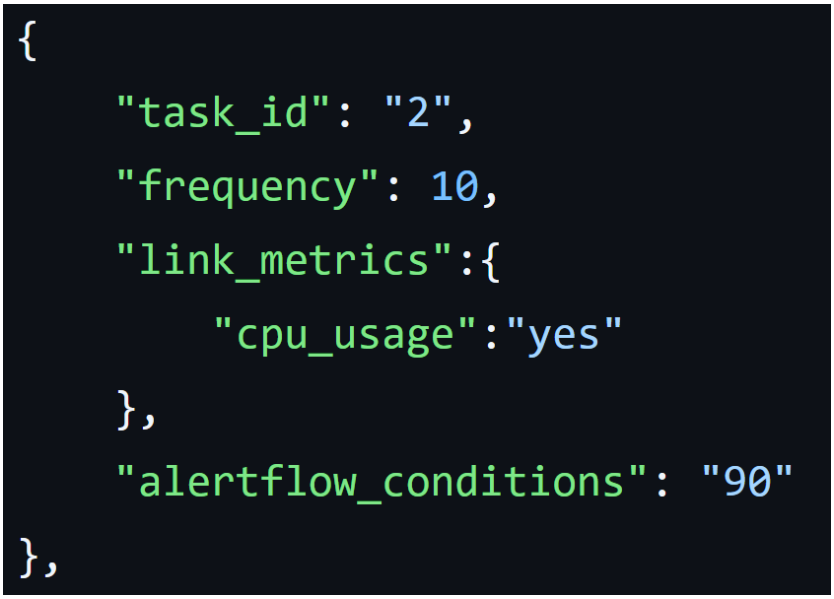

Figura 2 - Exemplo de tarefa de medição de CPU

Para a medição da percentagem de uso de CPU recorre-se ao comando “mpstat -P 0 <frequência>”, substituindo-se <frequência> pelo valor de *frequency* indicado na tarefa. Na apresentação do output, é necessário obter o valor correspondente a *idle* e subtraí-lo a 100.

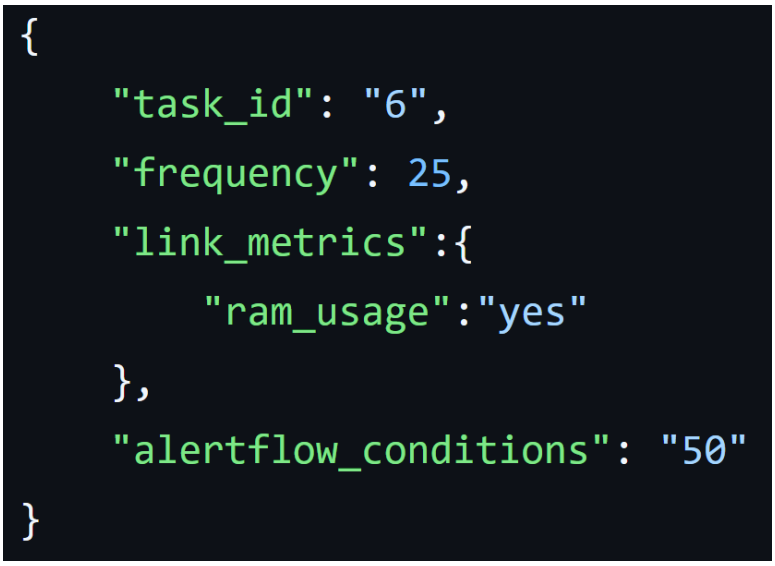

Figura 3 - Exemplo de tarefa de medição de RAM

Para a medição da percentagem de uso de RAM recorre-se ao comando "free -h -s <frequência>", substituindo-se, novamente, <frequência> pelo valor de *frequency* indicado na tarefa. Para a apresentação do output, é necessário obter o valor correspondente a *used*, dividi-lo pelo valor em *total* e multiplicar o resultado por 100, para obter uma percentagem.

Figura 4 - Exemplo de tarefa de medição de latência

Para a medição da latência recorre-se ao comando "ping -c <packets> -i <frequency_packets> <ip>", substituindo-se <packets> e <frequency_packets> pelos valores correspondentes indicados na tarefa e <ip> pelo valor em *destination*. Para apresentação do output, é necessário obter o valor correspondente a *avg*.

Figura 5 - Exemplos de tarefas de medição de largura de banda (esquerda), de perda de pacotes (centro) e de jitter (direita)

Para estas três tarefas, o comando usado é o mesmo: “iperf3 -s -i <frequência>”. Para o output, no caso da largura de banda, recorre-se ao campo *bitrate* do servidor e cliente *iperf*, para a perda de pacotes ao último campo (valor em percentagem) do output do servidor e para o jitter ao campo *jitter*.

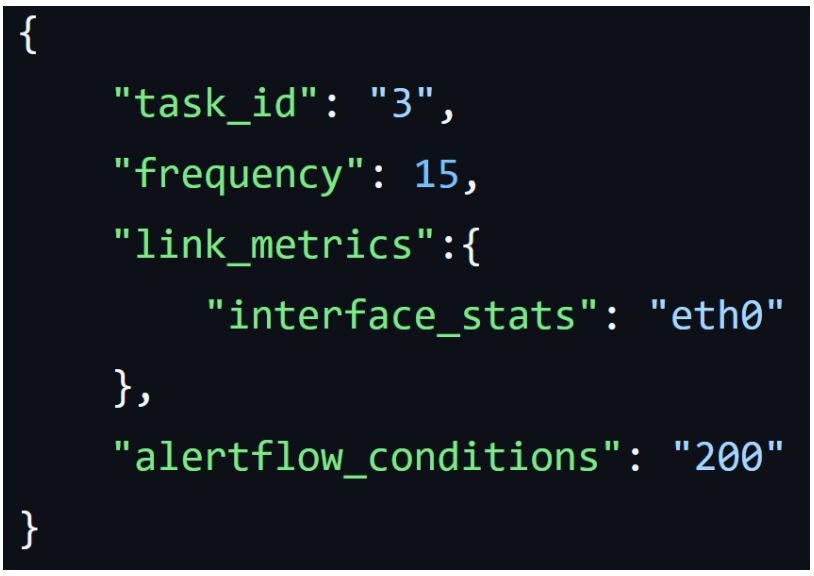

Figura 6 - Exemplo de tarefa para verificação do estado de uma interface

Por último, para obter o estado de uma interface recorre-se ao comando “ip link show eth<dados>”, substituindo-se <dados> pelo valor obtido de *interface_stats*. Para apresentação do output, é necessário obter a palavra que aparece a seguir a *state* (que pode ser UP ou DOWN).