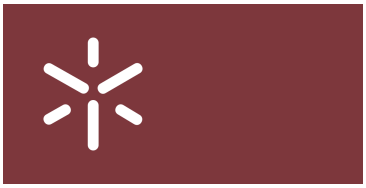

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Comunicações por Computador

Ano Letivo de 2024/2025

Monitorização Distribuída de Redes

Fábio Magalhães A104365
André Pinto A104267
Gonçalo Costa A93309

Dezembro, 2024

Data da Receção	
Responsável	
Avaliação	
Observações	

Monitorização Distribuída de Redes

Fábio Magalhães A104365
André Pinto A104267
Gonçalo Costa A93309

Dezembro, 2024

Resumo

O presente trabalho teve como objetivo desenvolver um sistema de monitorização distribuída de redes capaz de coletar métricas detalhadas sobre o desempenho de dispositivos e links, bem como notificar em tempo real sobre condições críticas.

O servidor central é responsável por coordenar os agentes, interpretar tarefas descritas em ficheiros JSON, e armazenar as métricas coletadas, utilizando os protocolos **NetTask** (baseado em UDP) e **AlertFlow** (baseado em TCP). Os agentes, por sua vez, executam tarefas de monitorização, como medições de latência, largura de banda e uso de recursos, reportando os dados ao servidor e enviando alertas em caso de anomalias. A modularidade do sistema foi assegurada por uma implementação organizada em ficheiros distintos, cobrindo funções específicas, como coleta de métricas, gestão de tarefas e comunicação.

O sistema foi testado em um ambiente simulado utilizando o emulador CORE, demonstrando eficácia na coleta e reporte de métricas, mesmo em cenários com múltiplos agentes. Limitações foram identificadas, como o controle simplificado de conexões no protocolo **AlertFlow**, que suporta até 5 conexões simultâneas sem uma lógica de redistribuição em casos de sobrecarga. No entanto, os resultados obtidos validam a confiabilidade e a escalabilidade do sistema para os cenários propostos.

Conclui-se que a solução desenvolvida cumpre os objetivos iniciais e apresenta um ponto de partida sólido para futuros aprimoramentos, como controle avançado de conexões, integração com plataformas de gestão de incidentes e suporte a um maior número de agentes.

Área de Aplicação: Desempenho e monitorização em Redes de Computadores

Palavras-Chave: Python, Camada de Transporte, Redes de Computadores, Socket, Desempenho

Índice

Lista de Figuras

Lista de Tabelas

1. Introdução

Cada vez mais os sistemas do nosso dia a dia dependem de redes de computadores, logo a capacidade de monitorar o desempenho dessas redes é essencial para evitar interrupções nos serviços e garantir uma operação contínua e eficiente. O monitoramento proativo possibilita a identificação e resolução de problemas antes que se tornem críticos, garantindo a satisfação dos usuários e a continuidade dos negócios.

O objetivo deste trabalho foi desenvolver um sistema de monitoramento distribuído de redes, que seja capaz de coletar métricas detalhadas sobre o desempenho de links e dispositivos conectados, além de alertar em tempo real sobre eventuais anomalias. Para atingir esse objetivo, será implementada uma solução baseada em uma arquitetura cliente-servidor, onde:

Um servidor centralizado será responsável por coordenar agentes distribuídos, interpretar tarefas a partir de arquivos JSON e armazenar as métricas coletadas. Agentes distribuídos executarão tarefas específicas, como medir latência, largura de banda e uso de recursos, reportando os dados ao servidor. Dois protocolos aplicacionais serão projetados para suportar essa comunicação:

- NetTask (UDP): Para a troca eficiente de tarefas e coleta contínua de métricas.
- AlertFlow (TCP): Para notificação de eventos críticos, garantindo entrega confiável das mensagens.

Este projeto, inspirado no protocolo NetFlow, também busca explorar conceitos fundamentais de comunicação por redes, como o uso de sockets, números de sequência, retransmissão e controle de fluxo, aplicados em um ambiente distribuído. O trabalho culminará em uma demonstração prática utilizando o emulador CORE para simular redes com diferentes condições de conectividade.

2. Arquitetura da solução

A solução desenvolvida para monitorização distribuída de redes baseia-se em uma arquitetura modular e escalável, composta por um servidor central e múltiplos agentes distribuídos que se comunicam através de dois protocolos específicos: **NetTask (UDP)** e **AlertFlow (TCP)**. A seguir, são detalhados os componentes principais.

2.1. Servidor Central (`NMS_Server.py`)

O servidor central é responsável por coordenar a comunicação com os agentes, interpretar tarefas descritas em ficheiros JSON, atribuí-las aos agentes e armazenar métricas recolhidas. Suas principais funcionalidades incluem:

- **Gestão de Tarefas:** Processamento de tarefas a partir do ficheiro de configuração JSON, utilizando o módulo `task.py`.
- **Armazenamento de Dados:** Receção e armazenamento das métricas enviadas pelos agentes.
- **Comunicação com os Agentes:** Realizada via UDP (**NetTask**) e TCP (**AlertFlow**).

2.2. Agentes Distribuídos (`NMS_Agent.py`)

Os agentes representam os dispositivos monitorados e são responsáveis por executar medições locais e reportar métricas ao servidor. Suas principais funções incluem:

- **Execução de Tarefas:** Coleta de métricas como uso de CPU, RAM, estatísticas de interface de rede e métricas de comunicação (e.g., latência, jitter, largura de banda e perda de pacotes) utilizando ferramentas como `ping` e `iperf3`.
- **Envio de Dados:** Reporte periódico das métricas ao servidor via **NetTask**.
- **Disparo de Alertas:** Notificação de eventos críticos ao servidor via **AlertFlow**.

2.3. Ficheiros Complementares

- **`main.py`:** Arquivo de execução principal, que permite iniciar tanto o servidor quanto os agentes.
- **`protocols.py`:** Define os protocolos **NetTask** e **AlertFlow**, incluindo a lógica para registar agentes, gerenciar tarefas e transmitir mensagens.
- **`metrics.py`:** Implementa as funções de coleta de métricas, como monitorização de CPU, RAM, estatísticas de interface de rede e métricas de link (latência, jitter, largura de banda e perda de pacotes).
- **`task.py`:** Gerencia as tarefas carregadas a partir do ficheiro JSON, além de associar-las aos dispositivos.
- **`notify.py`:** Implementa um sistema de notificações organizando mensagens em diferentes níveis de prioridade.
- **`encoder.py`:** Serializa e desserializa mensagens trocadas entre os agentes e o servidor.

2.4. Fluxo de Execução

1. **Registro do Agente:** Cada agente regista-se no servidor via **NetTask**, obtendo um identificador único.
2. **Envio de Tarefas:** O servidor distribui tarefas aos agentes, definindo quais métricas devem ser coletadas e com que frequência.
3. **Coleta e Reporte de Métricas:** Os agentes executam as tarefas e reportam os resultados ao servidor via **NetTask**.
4. **Disparo de Alertas:** Em caso de anomalias, os agentes notificam o servidor via **AlertFlow**.
5. **Armazenamento e Visualização:** O servidor armazena as métricas e fornece uma interface para visualização.

3. Especificação dos protocolos propostos

Os protocolos utilizados desempenham papéis complementares no sistema, sendo fundamentais para a comunicação entre o servidor e os agentes.

3.1. NetTask (UDP)

Este protocolo é usado para envio de tarefas do servidor para os agentes e para a recolha de métricas. Características principais:

- **Eficiência:** Utiliza UDP para minimizar overhead.
- **Confiabilidade adicional:** Inclui números de sequência, retransmissão e controle de fluxo inspirados no TCP.

3.2. AlertFlow (TCP)

Protocolo confiável utilizado para notificações de eventos críticos. Especificações:

- **Limite de Conexões:** Configurado para até **5 conexões simultâneas** no servidor.
- **Mensagens Estruturadas:** Informações sobre métricas excedidas e valores críticos são enviadas de forma confiável.
- **Limitações:** O controle detalhado de conexões não foi implementado, o que pode causar falhas em alta demanda.

3.3. Formato das mensagens protocolares

- **NetTask:** Estruturadas com informações sobre a tarefa, como ID, frequência e dispositivos alvo.
- **AlertFlow:** Contêm identificadores de métricas críticas excedidas e os valores detetados.

3.3.1. Design Específico

No caso do protocolo **AlertFlow**, foi estabelecido um limite de **5 conexões simultâneas** no servidor. Este limite foi configurado para garantir a estabilidade da aplicação e evitar sobrecarga no sistema, pois **não foi implementado um controle de conexão detalhado**, que contasse/expandisse as conexões quando estas atingissem o máximo esperado. Logo o nosso protocolo TCP apenas dará “listen” a 5 hosts.

3.4. Diagrama de sequência data troca de mensagens

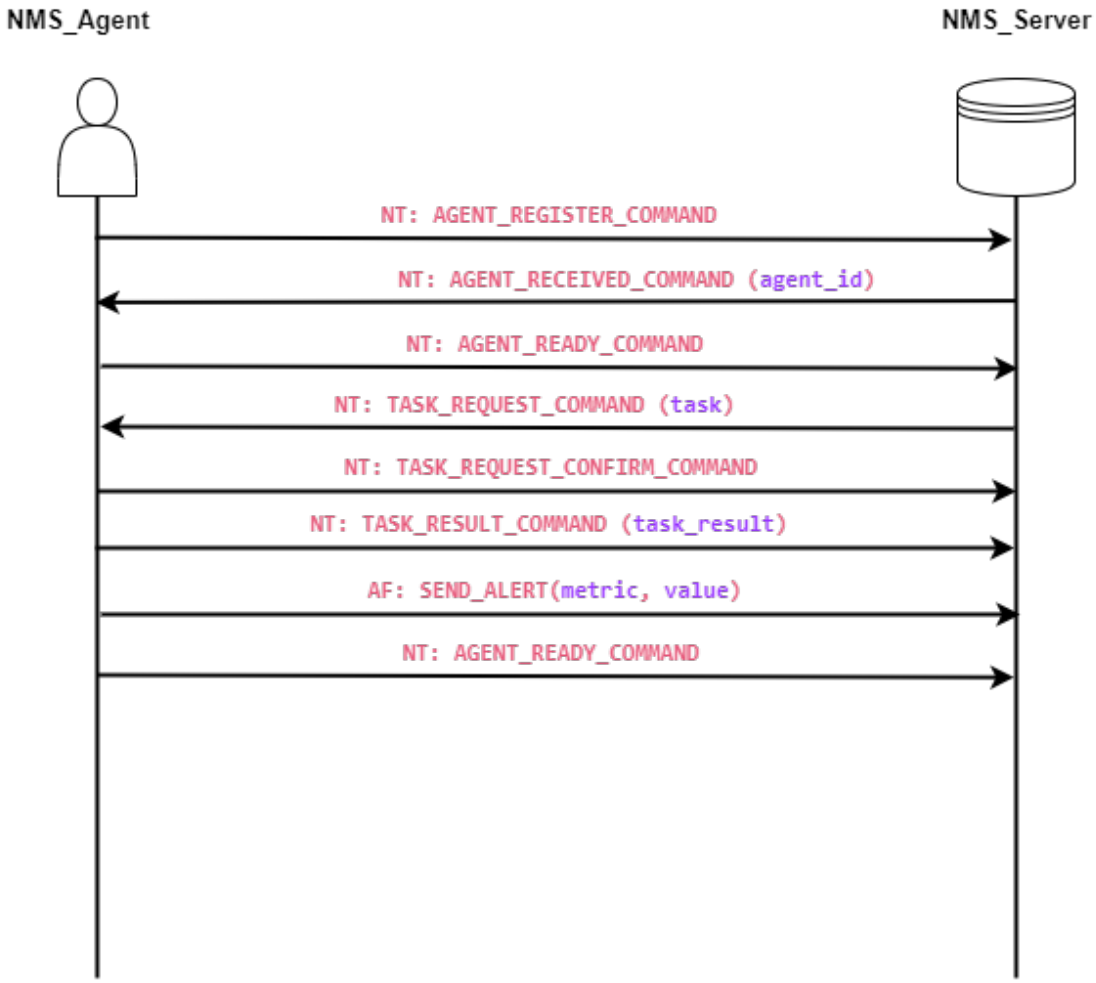

Figura 1: Diagrama temporal das interações entre servidor e agentes.

Neste diagrama temporal é possivel verificar a interação entre cliente e servidor de ambos os protocolos, no diagrama representados por **NT** como o `NetTask` e **AF** como o `AlertFlow`.

Também é possível neste mesmo diagrama verificar as mensagens de acknowledgment, que apenas confirmam a receção e mensagens com informação (representada abstratamente).

- agent_id: o id do agente que se registou
- task: o id da tarefa a ser executada
- task_result: o resultado da tarefa
- (metric, value): um tuplo com o nome da métrica que foi excedida e o valor excedente

4. Implementação

O projeto foi desenvolvido em **Python** devido à sua simplicidade e à disponibilidade de bibliotecas adequadas para manipular redes e processos. O sistema segue uma arquitetura modular com um servidor central que distribui tarefas e agentes distribuídos que coletam métricas e alertam sobre anomalias.

4.1. Detalhes Técnicos e Bibliotecas Utilizadas

Para implementar a comunicação entre o servidor e os agentes, bem como para garantir o processamento paralelo de tarefas e a coleta de métricas, foram utilizadas diversas bibliotecas e técnicas:

- **`socket`**: Utilizado para estabelecer a comunicação de rede, tanto com TCP (para AlertFlow) quanto com UDP (para NetTask).
- **`threading`**: Cada componente (servidor e agentes) opera em paralelo graças ao uso de threads. Exemplos incluem:
  - O servidor cria uma thread para cada agente que se conecta, mantendo assim a comunicação contínua com múltiplos agentes simultaneamente.
  - Os agentes, por sua vez, criam threads para executar as tarefas, recolher métricas e enviar resultados, garantindo que não haja bloqueios durante as operações.
- **`psutil`**: Biblioteca usada para recolher métricas de sistema, como o uso de CPU e memória. Isso facilita o monitoramento dos dispositivos pelos agentes.
- **`subprocess`**: Para executar comandos externos como `ping` e `iperf3`, que são utilizados na medição de latência e largura de banda dos links.

4.2. Fluxo de Execução dos Componentes

4.2.1. Inicialização e Registo dos Agentes

Quando o servidor é iniciado, ele começa a escutar em uma porta específica tanto para a comunicação UDP (**NetTask**) quanto para a comunicação TCP (**AlertFlow**). Os agentes, ao serem executados, registram-se no servidor enviando uma mensagem de "REGISTER" através do protocolo **NetTask**. O servidor então atribui um identificador único a cada agente.

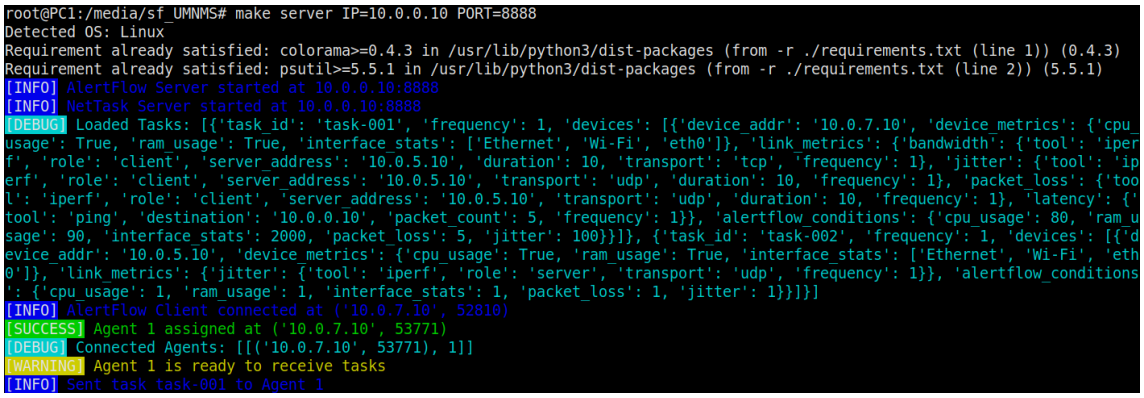

Figura 2: Mostra que o server é iniciado, lida as tasks, agente 1 liga-se, é lhe atribuido um id 1, agente 1 envia que está pronto, de seguida é enviado request de run task para o agente 1 e agente confirma e corre a task.

4.2.2. Execução de Tarefas e Coleta de Métricas

Após o registro, os agentes ficam prontos para receber as tarefas distribuídas pelo servidor. As tarefas são recebidas via UDP e podem envolver medições de uso de CPU, RAM, latência, e largura de banda.

- Cada tarefa recebida é executada em uma thread separada, o que permite que o agente processe várias tarefas de forma assíncrona.

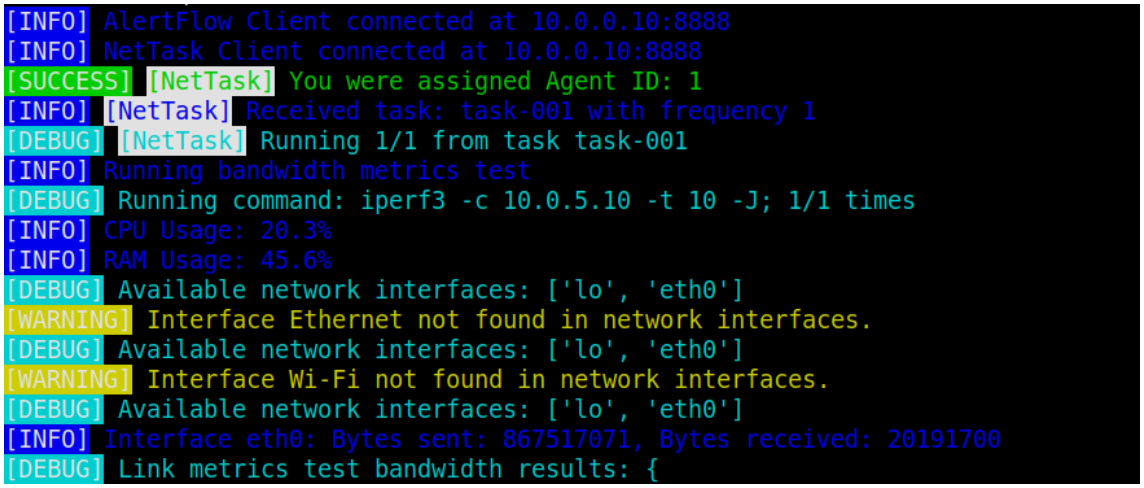

Figura 3: Mostra que Agente liga-se, recebe o seu id, envia o ready, recebe a task, e começa a correr a task (`[INFO] NetTask Received task: task-001 with frequency 1`).

Durante a execução, as métricas são recolhidas com o auxílio de `ping` e `iperf3`, e o uso de recursos é monitorado com `psutil`. Os valores das métricas, como o uso de CPU e a largura de banda, são coletados e enviados de volta ao servidor.

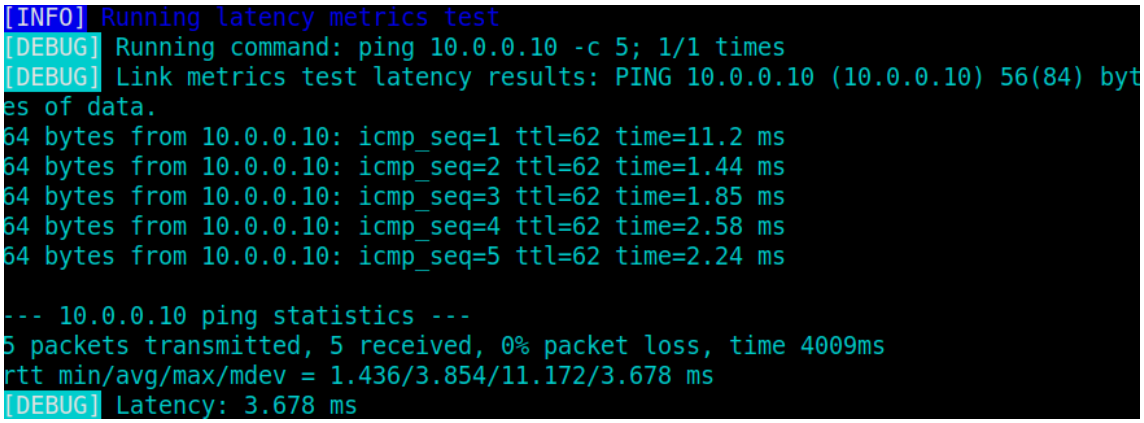

Figura 4: Mostra que Agente acaba de correr a task e envia que está pronto para correr tasks novamente e envia os alertas caso existam excedentes dos limites/condições definidas mais logs do agente durante a execução da tarefa, com detalhes de uso de CPU e RAM (`[INFO] CPU Usage: 20.3%`, `[INFO] RAM Usage: 45.6%`).

4.2.3. Geração de Alertas

Caso uma métrica exceda um limite predefinido (por exemplo, alto uso de CPU ou perda de pacotes significativa), o agente gera um alerta e envia essa notificação ao servidor através do protocolo **AlertFlow** (TCP). Isso permite uma comunicação confiável para eventos críticos.

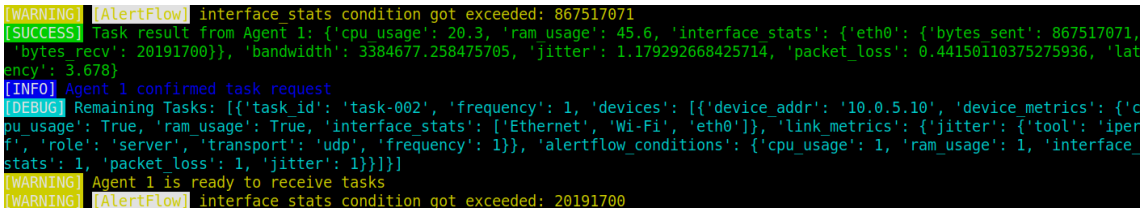

Figura 5: Mostra que Server recebe os resultados, captura o envio de alertas pelo agente ao detectar uma condição crítica (`[WARNING] [AlertFlow] interface_stats condition got exceeded`) e recebe informação que o agente está pronto. Posteriormente o agente verifica se há mais tasks e continua a correr.

4.2.4. Salvamento dos resultados

No final da execução da tarefa o seu resultado é guardado num ficheiro em formato JSON ou textual numa pasta predefinida (`data/outputs`).

4.3. Lógica de Comunicação e Manutenção da Conexão

A comunicação entre o servidor e os agentes ocorre de maneira contínua e confiável:

- **NetTask**: Responsável por enviar tarefas e receber resultados através de UDP. A lógica de retransmissão é utilizada caso uma resposta não seja recebida em um determinado período.
- **AlertFlow**: Garante que todas as notificações de eventos críticos sejam entregues, utilizando o TCP para assegurar a confiabilidade.

5. Testes e Resultados

Para validar a implementação do sistema de monitorização distribuída de redes, foram realizados diferentes tipos de testes com o objetivo de avaliar o desempenho e a eficiência da comunicação entre o servidor e os agentes, bem como a resposta a condições críticas.

5.1. Testes e Resultados Observados

- **Teste de Registro e Inicialização**: Verificar se todos os agentes são corretamente registrados e recebem seu ID. O servidor foi capaz de gerir múltiplos agentes simultaneamente e atribuir IDs únicos a cada um deles.

Resultado: Todos os agentes conseguiram se registrar corretamente no servidor, e as tarefas foram distribuídas sem falhas.

- **Teste de Execução de Tarefas**: Medir a eficiência do sistema na distribuição e execução de múltiplas tarefas simultaneamente. O uso de threads foi essencial para garantir que não houvesse bloqueios.

Resultado: A coleta de métricas foi realizada com sucesso, e os dados recolhidos foram enviados ao servidor conforme esperado.

- **Teste de Geração de Alertas**: Simular situações em que métricas ultrapassam os limites definidos e observar como o sistema responde em termos de notificações e registro de eventos. Os alertas foram recebidos de forma confiável pelo servidor e registrados conforme esperado.

Resultado: Os alertas foram gerados e entregues ao servidor quando as métricas ultrapassaram os limites, demonstrando a eficácia do sistema na resposta a eventos críticos.

- **Teste de Multiplos Hosts**: Simular vários hosts em simultâneo ligados a um mesmo Servidor.

Resultado: Todos os hosts foram conectados ao servidor da forma esperada e gerados ids únicos para cada um.

- **Teste de Perda de Agentes**: Foi simulado a perda de um agente, isto ajuda a simular casos em que o agente se desconecta ou ocorrência de erros na máquina de execução desse mesmo.

Resultado: O agente foi conectado de forma correta e forçado a desconectar “matando” o processo que o executava. O que se verificou é que foi atribuido, como esperado, o mesmo id de agente e a tarefa foi re-atribuida ao mesmo para ser executada.

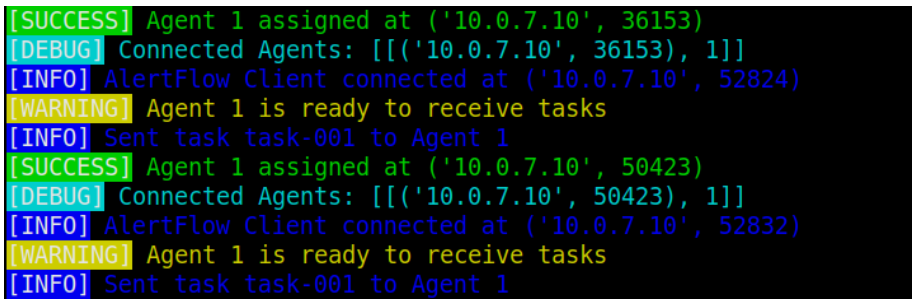

Figura 6: Host a ser re-conectado e atribuido o mesmo `agent id` e tasks.

- **Teste de Perda de Pacotes**: Simulado através do core a perda de pacotes e informação durante a conexão.

Resultado: Ocorrendo, apesar de raramente, a perda de informação, o programa verifica essa falta e volta a correr a task novamente. É importante notar que a falta de informação foi caracterizada pelo grupo como informação que foi recebida pelo comando executado mas não se encontra no resultado final; caso o comando não corra, por razões da própria máquina ou do processo, essa informação não vai constar no resultado final.

6. Conclusões

O desenvolvimento deste projeto demonstrou a viabilidade e a importância de um sistema distribuído de monitorização de redes, capaz de coletar métricas detalhadas e notificar em tempo real sobre condições críticas. A solução apresentou um bom desempenho nas operações de registo de agentes, distribuição de tarefas e comunicação confiável por meio dos protocolos **NetTask** e **AlertFlow**. Além disso, a modularidade implementada nos ficheiros facilita a manutenção e a escalabilidade do sistema.

Entretanto, algumas limitações foram identificadas, como o controle simplificado de conexões no protocolo **AlertFlow**, que pode causar falhas em cenários de alta demanda. Apesar disso, os testes realizados demonstraram que o sistema é eficiente para a coleta e processamento de métricas e para a emissão de alertas, atendendo aos requisitos definidos no início do projeto.

6.1. Trabalho Futuro

Para aprimorar a solução desenvolvida, algumas melhorias e extensões podem ser consideradas para o futuro:

- **Controle Avançado de Conexões:** Implementar uma lógica robusta para gerenciar conexões no protocolo **AlertFlow**, como filas de espera ou redistribuição de conexões quando o limite for excedido.
- **Análise de Desempenho:** Ampliar os testes realizados para incluir cenários mais complexos, como redes com alta latência e perda de pacotes, avaliando o impacto nos tempos de resposta e na confiabilidade do sistema.
- **Escalabilidade:** Explorar técnicas para suportar um número maior de agentes, como balanceamento de carga ou divisão dos agentes em clusters gerenciados por servidores dedicados.
- **Automação de Alertas:** Integrar o sistema com plataformas de gestão de incidentes, permitindo ações automáticas com base nos alertas gerados (e.g., reinício de serviços ou reconfiguração de dispositivos).

Com estas melhorias, espera-se que o sistema possa atender a um conjunto mais amplo de requisitos e se adaptar a diferentes cenários de operação, consolidando-se como uma solução confiável para a monitorização distribuída de redes.