

Ferramenta de Monitorização

Trabalho Prático 2 | Gestão de Redes

Mestrado Integrado em Engenharia de Telecomunicações e Informática

Leandro Henrique Dantas Alves A82157

Data de Entrega: 22/12/2019



Introdução

Neste trabalho prático pretende-se criar um programa para a monitorização dos processos ativos de um *host* e disponibilizar uma qualquer forma de visualização para que se possa monitorizar a evolução dos vários processos. Esta pode ser construída numa interface *web* para se poder utilizar a ferramenta de monitorização.

A Ferramenta

Backend

No *backend*, a ferramenta foi desenvolvida em linguagem *java*, com a implementação da biblioteca *snmp4j*. Esta permitiu que fosse facilmente criado um programa para a obtenção dos vários processos que estão em execução e, assim, monitorizar o *host* pretendido.

Para que o programa corresse devidamente foi necessário a criação de várias etapas/procedimentos.

As etapas adotadas foram as seguintes:

1. Configuração do cliente SNMP (host)

É criado um cliente SNMP através do endereço e da porta indicados pelo Utilizador na interface *web*, sendo também necessário uma *community String* e a versão em que o SNMP irá correr, estes dois últimos parâmetros estão definidos na aplicação.

2. Configuração das OIDs necessárias para a leitura da informação

Introdução das OIDs numa Lista. Estas OIDs estão explicadas mais à frente e o porquê da sua escolha [abaixo].

3. Leitura dos Dados

Nesta parte são obtidos dois tipos de dados: a informação sobre o *host* (número total de processos, tempo desde o arranque) e todos os processos ativos, assim como a utilização do CPU e da RAM.

Para obter a informação do *host* é utilizado o equivalente a um comando *snmpget* da biblioteca *snmp4j*. Para obter todos os processos ativos e os seus valores de CPU e RAM é utilizado o método *treeUtils* do *snmp4j* que percorre as várias OIDs que foram indicadas.

4. Separação dos processos que tem o maior consumo de CPU e RAM

Depois de se obterem os processos, procedimento [3], estes são filtrados e armazenados numa lista, para serem apresentados na interface *web*.

Interface *web* / *Frontend*

Na interface *web*, foi utilizado HTML para a visualização dos processos recolhidos no *backend*. Esta interface foi dividida em duas secções: uma para introdução dos dados do cliente a monitorizar e outra para a visualização dos diversos processos do cliente introduzido. Esta segunda secção foi separada em três componentes, para uma melhor compreensão e organização dos processos. Ainda nesta segunda secção, foi criada uma lista com algumas informações (o endereço, a porta, o tempo ativo e o número total de processos) de cada cliente que o utilizador escolheu monitorizar.

Na primeira secção, o Utilizador introduz o endereço e a porta do cliente que quer monitorizar, e caso pretenda, pode ainda seleccionar um tempo de atualização dos dados. O tempo de atualização tem, por defeito, o valor de 1 minuto.

Como já foi dito, a segunda secção está dividida em três componentes. Os componentes de visualização são os seguintes:

- **Cartões de Informação**
 - **Dados do Cliente:** Indicação o endereço e a porta do *host*.
 - **Tempos:** Indicação do tempo desde que o *host* arrancou e o tempo até à próxima atualização dos dados.
 - **Número total de processos:** Indicação do número dos processos ativos.
- **Gráficos de maior consumo**
 - **Consumo de CPU:** Apresentação dos consumos de CPU dos vários processos.
 - **Consumo de RAM:** Apresentação dos consumos de memória dos vários processos.
- **Tabela com todos os processos ativos**

Para cada processo são apresentadas as seguintes variáveis:

 - **Nome do processo**
 - **Consumo de CPU**
 - **Consumo de RAM**
 - **Estado do processo**

Manual de Instruções

Quando o utilizador aceder à interface *web*, vão-lhe ser pedidas 3 informações: o endereço do cliente, a porta e o tempo de atualização de dados. Após a introdução destas informações, basta apenas clicar no botão de "Monitorizar". Em seguida, o utilizador será encaminhado para uma página onde poderá visualizar os processos do cliente introduzido.

Escolha dos Objetos da MIB

Como foi dito acima, foram escolhidos vários objetos da MIB, **HOST-RESOURCES-MIB**, para que a ferramenta funcionasse de modo a apresentar todos os valores necessários na interface *web*. Para isso foram escolhidos os seguintes objetos:

- **hrSystemUptime**: Usado para indicar o tempo desde que o *host* arrancou.
- **hrSystemProcesses**: Usado para indicar o número total de processos ativos.
- **hrSWRunIndex**: Usado para distinguir os processos no histórico.
- **hrMemorySize**: Valor da memória RAM usado para efetuar cálculos.
- **hrSWRunStatus**: Usado para indicar o estado do processo.
- **hrSWRunName**: Usado para se saber qual é o nome do processo.
- **hrSWRunPerfCPU**: Consumo de CPU que o processo está a gastar.
- **hrSWRunPerfMem**: Consumo de RAM que o processo está a gastar.

Só foram escolhidos estes objetos, pois são suficientes para monitorizar os vários clientes, uma vez que indicam os consumos de CPU e RAM que os diversos processos estão a gastar, assim como se tal processo está ativo ou não. Alguns destes valores são recolhidos para cálculos, como o caso de **hrSystemUptime** e **hrMemorySize**.

O **hrSystemUptime**, não só é utilizado para apresentar o tempo, mas também para calcular a percentagem de CPU que o processo está a gastar. O cálculo que é efetuado é o seguinte:

$$\% \text{ CPU Utilizado por processo} = \frac{hrSWRunPerfCPU}{hrSystemUptime}$$

O **hrMemorySize**, é obtido para se calcular a percentagem que cada processo está a utilizar. A seguinte fórmula demonstra como é calculada essa percentagem.

$$\% \text{ RAM utilizada por processo} = \frac{hrSWRunPerfMem}{hrMemorySize}$$

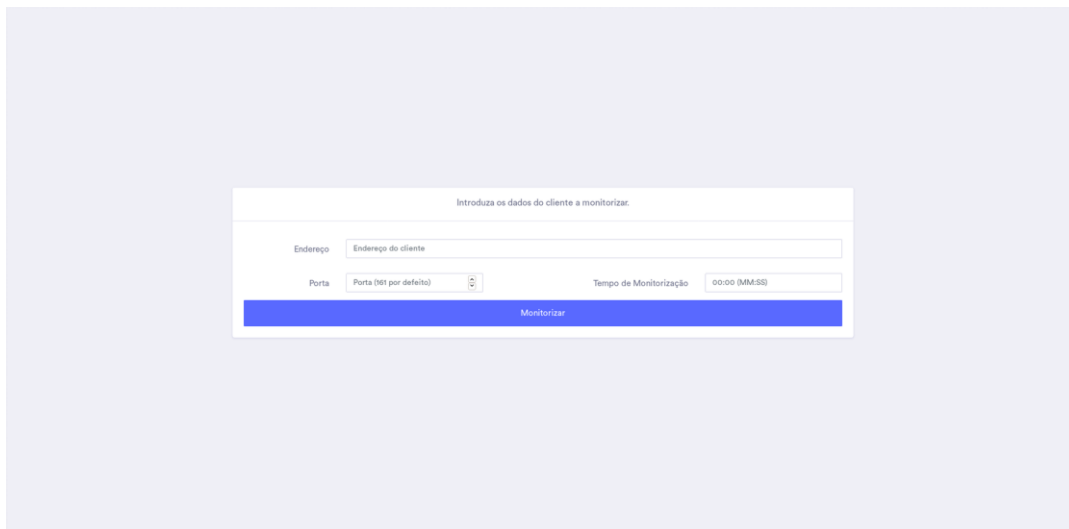
Tempos de Monitorização

Como já foi referido acima, o utilizador pode, se pretender, monitorizar um cliente por um tempo definido por ele, pois cada cliente deve ter o seu tempo de monitorização para uma melhor perspetiva sobre como os processos estão se a comportar.

Por defeito, este tempo tem um valor de 1 minuto, ou seja, caso o utilizador não quera estabelecer nenhum tempo de monitorização para o cliente. Foi escolhido 1 minuto, pois é tempo mais que suficiente para tirar conclusões sobre o estado dos processos.

Anexos

Como exemplo da utilização da aplicação de monitorização criada, foi utilizado, maioritariamente, o próprio computador, *localhost*, como se pode ver nos seguintes *snapshots* da interface *web*.



The screenshot shows a web form titled "Introduza os dados do cliente a monitorizar." (Introduce the data of the client to monitor). The form contains three input fields: "Endereço" (Address) with the placeholder "Endereço do cliente", "Porta" (Port) with a dropdown menu showing "Porta (881 por default)", and "Tempo de Monitorização" (Monitoring Time) with a time picker set to "00:00 (MM:SS)". A blue "Monitorizar" button is at the bottom of the form.

FIGURA 1: PÁGINA ONDE O UTILIZADOR INTRODUZ OS DADOS DO CLIENTE A MONITORIZAR.

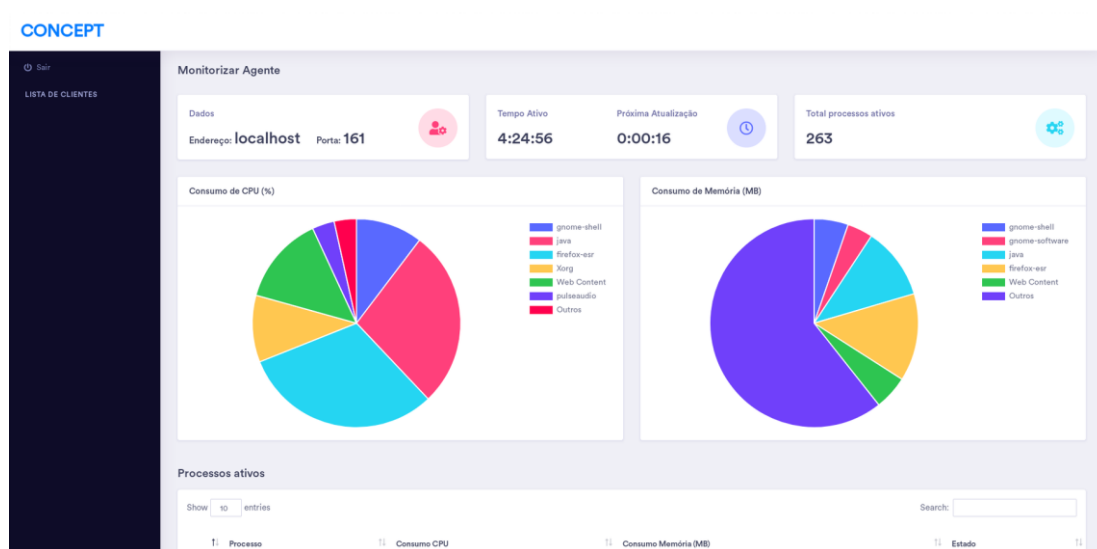


FIGURA 2: PÁGINA ONDE O UTILIZADOR OBSERVA OS VÁRIOS PROCESSOS E O QUE CADA UM CONSUME.

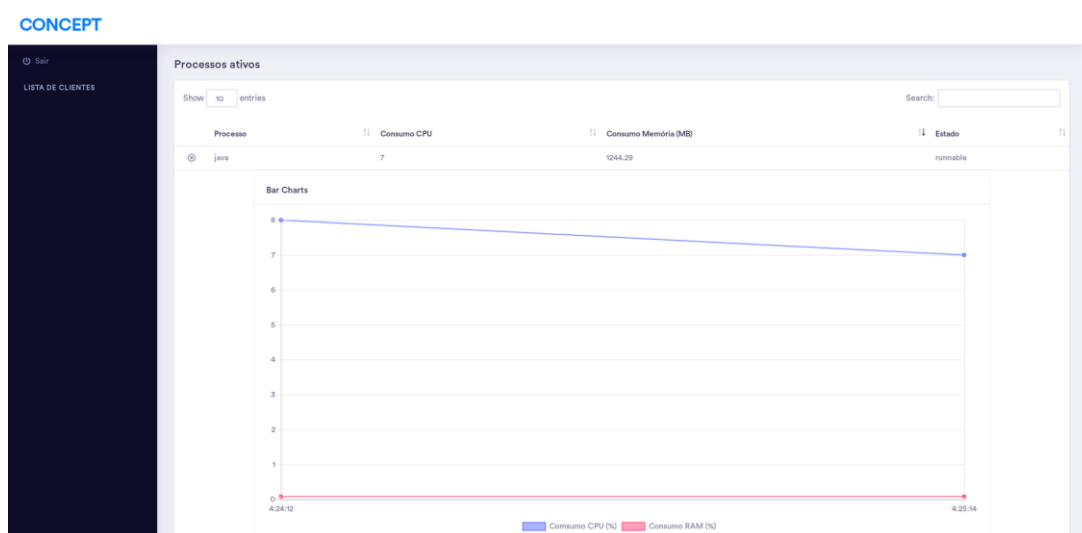


FIGURA 3: EXEMPLO DO HISTÓRIO DE UM PROCESSO

Código

Aqui só está apresentado o código java relativo ao *backend*. O projeto com ambos os códigos está, como anexo, junto ao e-mail enviado.

```
public class SNMPManagerService {

    Snmp snmp = null;
    String address = null;
    String port = null;
    String nextUpdate = null;
    ArrayList<OID> oids = new ArrayList<OID>(){
        add(new OID(".1.3.6.1.2.1.25.1.6.0"));
        add(new OID(".1.3.6.1.2.1.25.1.1.0"));
        add(new OID(".1.3.6.1.2.1.25.4.2.1.2"));
        add(new OID(".1.3.6.1.2.1.25.5.1.1.1"));
        add(new OID(".1.3.6.1.2.1.25.5.1.1.2"));
        add(new OID(".1.3.6.1.2.1.25.4.2.1.7"));
        add(new OID(".1.3.6.1.2.1.25.4.2.1.1"));
        add(new OID(".1.3.6.1.2.1.25.2.2.0"));

    });
    int ramSize = 0;
    String sysUpTime = "";
    HashMap<Integer,List> processes = new HashMap<>();
    HashMap<Integer,List> historyProcs = new HashMap<>();
    final String[] status = new
String[]{"","running","runnable","notRunnable","invalid"};

    public SNMPManagerService(final String add,final String p, final String
n) {
        address = add;
        port = p;
        nextUpdate = n;
    }

    public void start() throws IOException {

        final TransportMapping transport = new
DefaultUdpTransportMapping();
        snmp = new Snmp(transport);

        // Do not forget this line!

        transport.listen();

    }

}
```



```

public void getRAMSize() throws IOException {
    final PDU pdu = new PDU();
    final ResponseEvent event;
    pdu.add(new VariableBinding(getOids().get(7)));
    pdu.setType(PDU.GET);
    event = snmp.send(pdu, getTarget(), null);
    double number =
Double.parseDouble(event.getResponse().get(0).getVariable().toString());
    this.ramSize = (int) (int) number/1024;
}
    public int getNumberProcesses() throws IOException {
        final PDU pdu = new PDU();
        final ResponseEvent event;
        pdu.add(new VariableBinding(getOids().get(0)));
        pdu.setType(PDU.GET);
        event = snmp.send(pdu, getTarget(), null);
        int number =
Integer.parseInt(event.getResponse().get(0).getVariable().toString());
        return number;
    }
    public HashMap<String, String> getCpuUsage() {
        HashMap<String,String> mostCPUUsage = new HashMap<>();
        int newIndex = 0;
        int lastCpuUsage = 0;
        for (Integer key : this.processes.keySet().stream()
            .sorted((o1,o2) ->

Integer.compare(Integer.parseInt(this.processes.get(o2).get(1).toString()),
Integer.parseInt(this.processes.get(o1).get(1).toString()))
        )
            .collect(Collectors.toList())){
            if(newIndex < 8){
                int CPUUsage =
Integer.parseInt(this.processes.get(key).get(1).toString());

                if(mostCPUUsage.get(this.processes.get(key).get(0).toString()) != null){

                    int last =
Integer.parseInt(mostCPUUsage.get(this.processes.get(key).get(0).toString())
                );

                    int total = last + CPUUsage;

                mostCPUUsage.put(this.processes.get(key).get(0).toString(),
                Double.toString(total));

            }else{

```

```

mostCPUUsage.put(this.processes.get(key).get(0).toString(),
Integer.toString(CPUUsage));
    }

    newIndex++;
} else {
    lastCpuUsage
+=Integer.parseInt(this.processes.get(key).get(1).toString());
    mostCPUUsage.put("Outros", Integer.toString(lastCpuUsage));
}
}
//System.out.println("CPU "+ mostCPUUsage);

return mostCPUUsage;
}
public HashMap<String, String> getRamUsage() {
    HashMap<String,String> mostRAMUsage = new HashMap<>();
    int newIndex = 0;
    double lastRAMUsage = 0;
    for (Integer key : this.processes.keySet().stream()
        .sorted((o1,o2) ->

Double.compare(Double.parseDouble(this.processes.get(o2).get(2).toString()),
Double.parseDouble(this.processes.get(o1).get(2).toString()))
    )
        .collect(Collectors.toList())){
        if(newIndex < 10){
            double RAMUsage =
Double.parseDouble(this.processes.get(key).get(2).toString())/1024;

            if(mostRAMUsage.get(this.processes.get(key).get(0).toString()) != null){
                double last =
Double.parseDouble(mostRAMUsage.get(this.processes.get(key).get(0).toString(
))));

                double total = last + RAMUsage;

mostRAMUsage.put(this.processes.get(key).get(0).toString(),
Double.toString(total));

            } else {

mostRAMUsage.put(this.processes.get(key).get(0).toString(),
Double.toString(RAMUsage));
            }

            newIndex++;
        } else {

```

```

        lastRAMUsage
+=Double.parseDouble(this.processes.get(key).get(2).toString())/1024;
        mostRAMUsage.put("Outros", Double.toString(lastRAMUsage));
    }
}
//System.out.println("RAM "+ mostRAMUsage);
return mostRAMUsage;
}
public HashMap getHistory() {
    return this.historyProcs;
}
public Variable getSysUpTime() throws IOException {
    final PDU pdu = new PDU();
    final ResponseEvent event;
    pdu.add(new VariableBinding(getOids().get(1)));
    pdu.setType(PDU.GET);
    event = snmp.send(pdu, getTarget(), null);
    return event.getResponse().get(0).getVariable();
}
public HashMap getProcessesTable(int max) throws IOException {
    List<TreeEvent> l = null;
    TreeUtils treeUtils = new TreeUtils(snmp, new DefaultPDUFactory());
    OID[] rootOIDs = new OID[5];
    for (int i = 2; i<oids.size()-1; i++) {
        rootOIDs[i-2] = oids.get(i);
    }
    l = treeUtils.walk(getTarget(), rootOIDs);
    int index = 0;

    long time = this.getSysUpTime().toLong();

    for(TreeEvent t : l){
        List<String> temp = new ArrayList<>();
        List<String[]> tempHistory = new ArrayList<>();
        VariableBinding[] vbs= t.getVariableBindings();
        if ((vbs != null) && (vbs.length != 0)){
            String[] tempArr = new String[3];
            int pidProc =
Integer.parseInt(vbs[4].getVariable().toString());

            for(int i = 0; i<vbs.length;i++){
                if( i == 3 ){

temp.add(this.status[Integer.parseInt(vbs[i].getVariable().toString())]);
                }else if( i == 1 ){

```

```

temp.add(Long.toString((long)((float)vbs[i].getVariable().toLong()/time*100));
        }else{
            temp.add(vbs[i].getVariable().toString());
        }
    }
    if(historyProcs.get(pidProc) == null){
        tempArr[0] = temp.get(1);
        tempArr[1] =
Double.toString(Double.parseDouble(temp.get(2).toString())/(1024*this.ramSize)
);
        tempArr[2] = this.getSysUpTime().toString().replace(".",
"-").split("-")[0];
        tempHistory.add(tempArr);
        historyProcs.put(pidProc, tempHistory);
    }else{
        tempHistory = historyProcs.get(pidProc);
        tempArr[0] = temp.get(1);
        tempArr[1] =
Double.toString(Double.parseDouble(temp.get(2).toString())/(1024*this.ramSize)
);
        tempArr[2] = this.getSysUpTime().toString().replace(".",
"-").split("-")[0];
        historyProcs.get(pidProc).add(tempArr);

    }
    processes.put(index, temp);
    index++;
}

}

//System.out.println("History: "+this.historyProcs);
//System.out.println("procs: "+processes);
return processes;
}

public void setOids(String[] oid) {
    for (String newOid : oid) {
        oids.add(new OID(newOid));
    }
}

public String getAddress(){
    return this.address;
}

public String getPort(){
    return this.port;
}
}

```

```

public ArrayList<OID> getOids() {
    return oids;
}

/**
 * This method returns a Target, which contains information about where
the data
 * should be fetched and how. * @return
 */
private Target getTarget() {
    final Address targetAddress =
GenericAddress.parse("udp:"+address+"/"+port);
    final CommunityTarget target = new CommunityTarget();

    target.setCommunity(new OctetString("public"));

    target.setAddress(targetAddress);

    target.setRetries(2);

    target.setTimeout(1500);

    target.setVersion(SnmpConstants.version2c);

    target.setMaxSizeRequestPDU(65536);

    return target;
}
}

```

Conclusão

No princípio deste trabalho prático surgiram algumas dúvidas, pois como este era desenvolvido em *java* e era necessário criar uma interface *web* para apresentar os dados que do *java*, havia um certo desconforto em perceber como se interligaram ambos.

Outro problema encontrado, este em relação à biblioteca *snmp4j*, foi que esta não permitia executar pedidos do tipo *getbulk*, porque apenas devolvia 100 linhas, o que não é o ideal pois um *host* pode ter mais do que 100 processos ativos.

Contudo, estes dois problemas foram resolvidos, o primeiro com a ajuda dos professores, por indicarem várias opções de *java application servers*, tais como o *TomCat*, *Jetty* e *JBoss*. Quando ao segundo problema, foi resolvido aplicando um outro método da biblioteca utilizada. No final, acho que ficou tudo a correr como foi inicialmente pensado.