Creation of a Network Management System

André Ávila
up202006767

Matilde Silva
up202007928

I. Introduction

Many institutions struggle to find their footing when it comes to networking. Often a challenging and expensive task, the creation and management of a network system is necessary to the flourishing and evolution of an independent company. To address this difficulty we set our goal on creating a network management service. Through virtualization, more forethought is given to the networks. Planning becomes action and problems are encountered sooner, thus cutting down on cost and resources.

II. Related work

Some tools of network management are already used and well-established in the realm of networks. For example, Solar Winds[3] is a comprehensive network management solution known for its extensive monitoring and diagnostic capabilities. However, most of the work and inspiration for this project came from the practical classes of the Networks and Systems Management class[1]. Based on the laboratories, we knew what instances to create, what features to include and, most importantly, how to manage the created networks.

III. Solution

We decided to create a network management system, to facilitate visualization and virtualization of networks. The idealized prototype would have an accessible and user-friendly interface, with which specifications and configurations of docker containers would be created, and afterwards executed.

A. Technologies

First, the chosen technology. We decided to use Tauri[4], an open-source software framework designed to create cross-platform desktop applications. A big appeal of the technology is its wide compatibility. Tauri supports cross-platform application window creation (TAO) and WebView rendering (WRY), which allows for use across macOS, Linux and Windows platforms.

Additionally, Tauri is built using Rust[2], a programming language emphasizing performance, type safety, and memory safety. Lastly, Tauri's toolkit also includes essential tooling such as bundlers, CLI interfaces, and scaffolding kits.[5] For our project, we opted for a SolidJS with Typescript frontend, as Tauri does offer a wide range of options. The Tauri application frontend (Typescript) communicates with the backend (Rust) to perform whatever tasks the user chooses.

B. App usage

For details on how to run the app, please refer to the repository ReadMe.

After the app's startup is finished, the user is confronted with the home page, in which the user's configurations are showed. If no configurations exist, the user may create a new one. All configurations are stored in the user's computer under the directory `/home/user/.local/share/com.netking.dev/`. Each configuration has its own TOML file. It should be highlighted, different configurations do not interact with each other whatsoever. They are atomic.

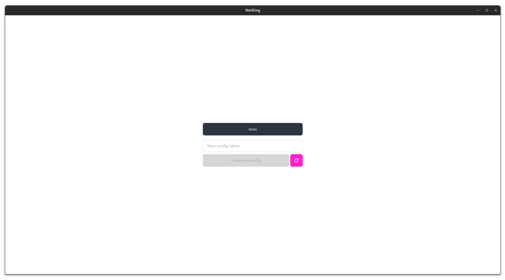
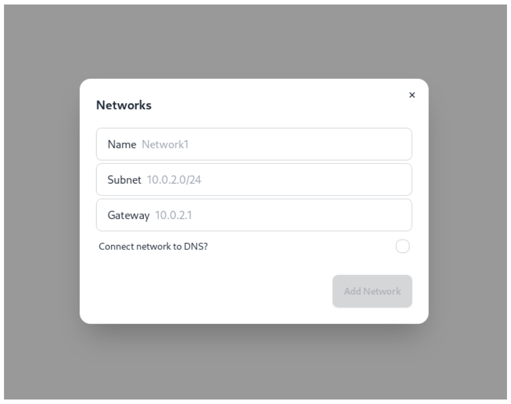


Fig. 1. Application's Landing Page

Upon selecting a configuration the user is shown its main page. In it, all configuration related operations occur. Creation or edition of instances or networks, container execution or halting and observation of relevant statistics.

C. Network and Instances Configuration

The user can create a network by specifying its name, subnet and gateway. Additionally, a checkbox is shown to give the option to connect the new network to the DNS network, but more on that later. The network is only created if the name doesn't already exist, if the given subnet is not already part of another network and if the gateway is part of the specified subnet. If all requirements are met, the network specification is written to the configuration's, previously mentioned, TOML file.

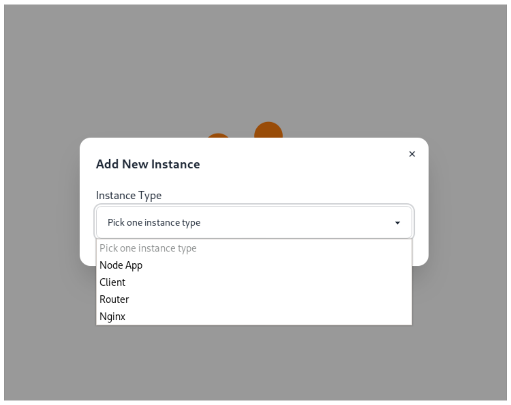


Name Network1
Subnet 10.0.2.0/24
Gateway 10.0.2.1
Connect network to DNS?

Fig. 2. Networks creation modal

Next, the user can also create container instances, which can be of the following type:

- NodeApp
- Client
- Router
- Nginx



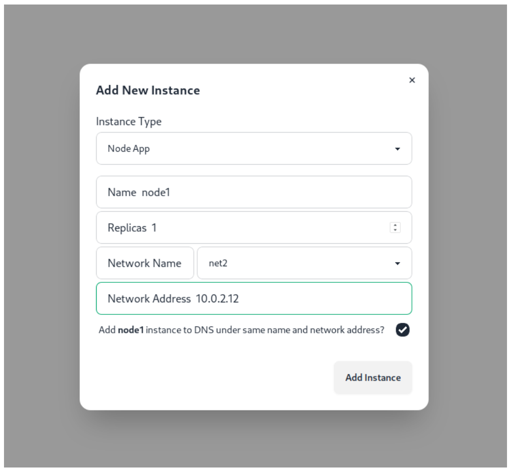
Instance Type
Pick one instance type

Fig. 3. Instances Creation Modal

If the NodeApp instance is selected, the NodeApp modal is shown. In it, the user specifies the name, number of replicas and the network to which it will belong to (chosen from a list of the configuration's existing networks). If the user does select a network, an additional field is revealed, in which the user may specify the NodeApp's IP address. In this case, a checkbox with the prompt ”Add instance to DNS under same name and network address?” is also shown.

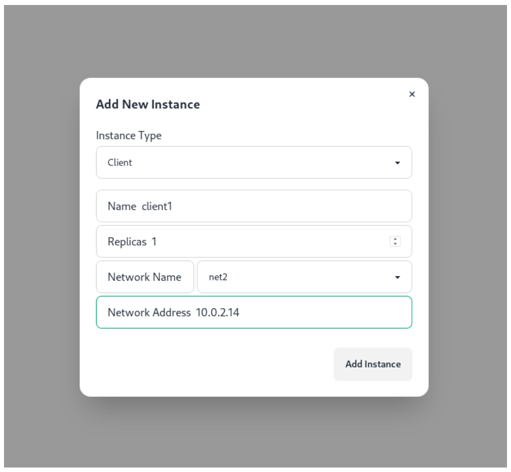
If the Client instance is selected, the Client modal is shown. It has the same fields and behaviour as the NodeApp modal, with the exception of the DNS checkbox.

If the Router instance is selected the Router modal is shown. The user can specify the name of the router instance and the network to which it will belong to. After selecting the network, the user must specify the IP address of the router instance. Unlike with the NodeApp and Client instances, all Router instances must have an associated network and IP address.



Instance Type
Node App
Name node1
Replicas 1
Network Name net2
Network Address 10.0.2.12
Add **node1** instance to DNS under same name and network address? ✓

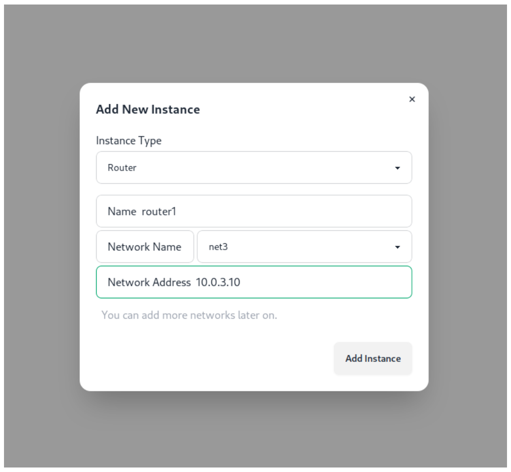
Fig. 4. NodeApp instance creation modal, with all fields filled in



Instance Type
Client
Name client1
Replicas 1
Network Name net2
Network Address 10.0.2.14

Fig. 5. Client instance creation modal, with all fields filled in

In all cases, however, the IP address needs to, obviously, be part of the selected subnet.

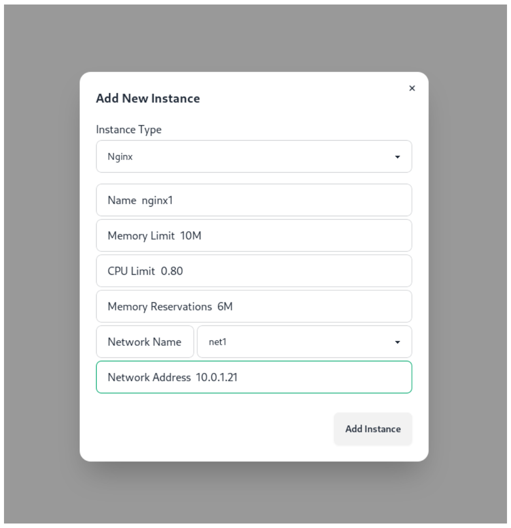


Instance Type
Router
Name router1
Network Name net3
Network Address 10.0.3.10
You can add more networks later on.

Fig. 6. Router instance creation modal, with all fields filled in

Lastly, if the Nginx instance is selected, the Nginx modal is shown. The user fills in the following fields:

- Name
- Memory Limit
- CPU Limit
- Memory Reservations
- Network Name (optional)
- IP Address (optional, must be part of the chosen network's subnet)


Add New Instance

Instance Type: Nginx

Name: nginx1

Memory Limit: 10M

CPU Limit: 0.80

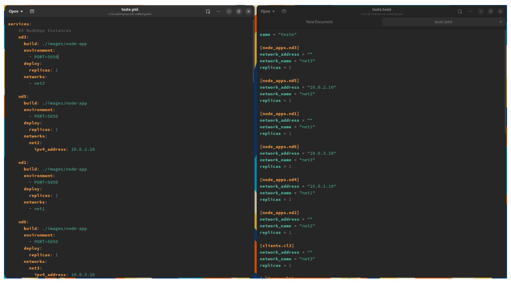
Memory Reservations: 6M

Network Name: net1

Network Address: 10.0.1.21

Fig. 7. Nginx instance creation modal, with all fields filled in

Anytime any of these instances are created, the configuration's TOML file is updated and stores the information the user has given. Whenever the TOML file is updated, a translation of it is made to YAML format. This YAML file is then used to create and run the Docker Containers.


Open
teste.yml
~/.local/share/com.netking.dev
Open
teste.toml
~/.local/share/com.netking.dev
New Document
teste.toml

```yaml
services:
  ## NodeApp Instances
  nd3:
    build: ./images/node-app
    environment:
      - PORT=5050
    deploy:
      replicas: 1
    networks:
      - net3

  nd5:
    build: ./images/node-app
    environment:
      - PORT=5050
    deploy:
      replicas: 1
    networks:
      net2:
        ipv4_address: 10.0.2.10

  nd1:
    build: ./images/node-app
    environment:
      - PORT=5050
    deploy:
      replicas: 1
    networks:
      - net1

  nd6:
    build: ./images/node-app
    environment:
      - PORT=5050
    deploy:
      replicas: 1
    networks:
      net3:
        ipv4_address: 10.0.3.10
```

```toml
name = "teste"

[node_apps.nd3]
network_address = ""
network_name = "net3"
replicas = 1

[node_apps.nd5]
network_address = "10.0.2.10"
network_name = "net2"
replicas = 1

[node_apps.nd1]
network_address = ""
network_name = "net1"
replicas = 1

[node_apps.nd6]
network_address = "10.0.3.10"
network_name = "net3"
replicas = 1

[node_apps.nd4]
network_address = "10.0.1.10"
network_name = "net1"
replicas = 1

[node_apps.nd2]
network_address = ""
network_name = "net2"
replicas = 1

[clients.cl3]
network_address = ""
network_name = "net3"
replicas = 1
```

Fig. 8. Example TOML and YML files

D. DNS

Anytime a new configuration is created, a DNS network for the configuration is simultaneously created too. The DNS subnet is always 192.168.1.0/30, no matter the configuration. The DNS resolver is also a fixed IP address, 192.168.1.2. The user can never create a network that contains the 192.168.1.0/30 subnet.

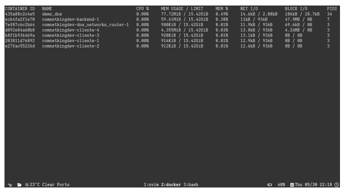
Furthermore, a router instance is also created. This router instance connects the DNS network and all other networks the users choosed to connect. This ensures communication between the user created networks and the DNS network.
Our DNS is supported by BIND, which performs both of the main DNS server roles, acting as an authoritative name server for DNS zones and as a recursive resolver in the network[6]. Two configuration files, `dns.<config_name>.net` and `<config_name>.conf.local` are created for this purpose. Whenever the user chooses to add a NodeApp instance to the DNS, the desired entry is written to the `dns.<config_name>.net` file.
The referred configuration files are then exported to the docker container, which automatically uses our DNS as the correct name resolver.

E. Container Execution and Statistics

As stated before, all configurations have an associated YAML file. The Rust back-end calls on the terminal and executes the command up with the flags ”-d”, ”–remove-orphans”, ”–force-recreate”, ”–no-log-prefix” and ”–quiet-pull”, each guarantee, respectively, that the containers run in detached mode, orphan containers are removed, containers are recreated and the logs shown are simplified. These flags were chosen to avoid conflicts with existing containers.

Docker Containers inherently have statistics that can be accessed using the docker stats command. We make use of this feature in the application. By running the command ”docker compose -f [the name of the configuration].yml stats –format json” in a background process, statistics from every container involved in the configuration are shown in the JSON format, which can easily be parsed. If they were not exported to JSON, the data that would otherwise be shown in the terminal (as seen in Figure 9) is shown in the NetKing UI (as seen in Figure 10).

The statistics for all running instances can be analyzed.



```
CONTAINER ID   NAME                                  CPU %     MEM USAGE / LIMIT     MEM %     NET I/O             BLOCK I/O         PIDS
435a88c2c4e5   demo_dns                              0.00%     77.72MiB / 15.42GiB   0.49%     14.6kB / 2.08kB     106kB / 28.7kB    34
ecb4fa2f1e70   comnetkingdev-backend-1               0.00%     59.61MiB / 15.42GiB   0.38%     11kB / 936B         47.9MB / 0B       7
7ef07c6c2b64   comnetkingdev-dns_networks_router-1   0.00%     980KiB / 15.42GiB     0.01%     11.9kB / 936B       69.6kB / 0B       3
d092e84a60b8   comnetkingdev-cliente-4               0.00%     4.355MiB / 15.42GiB   0.03%     13.8kB / 936B       4.26MB / 0B       3
b8f1b9364b9a   comnetkingdev-cliente-3               0.00%     920KiB / 15.42GiB     0.01%     13.1kB / 936B       0B / 0B           3
203811d76892   comnetkingdev-cliente-1               0.00%     916KiB / 15.42GiB     0.01%     12.9kB / 936B       0B / 0B           3
e27fac55226d   comnetkingdev-cliente-2               0.00%     912KiB / 15.42GiB     0.01%     12.6kB / 936B       0B / 0B           3
```

Fig. 9. Container stats in the terminal

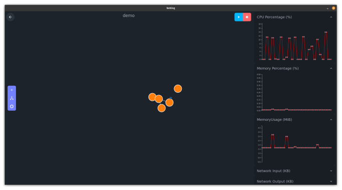


Fig. 10. Container stats in the NetKing UI

IV. Evaluation

A. Setup

In order to test a network created by the application, a configuration should be created first, then follows the creation of some instances.

In this case we will create a network called ”network1” with a subnet of ”10.0.1.0/24” and a gateway of ”10.0.1.1”. This network will be connected to the DNS through endpoint ”10.0.1.5” (see figure 11).

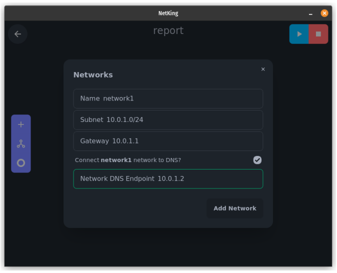
After this, multiple instances are created.

First, an instance of the type ”Node App”, named ”backend”, which will be a container running a Node application. This instance will be created as part of ”network1” with the IP address ”10.0.1.3”. Furthermore, the instance will be connected to the DNS.

Secondly, an instance of the type ”Client”, named ”clients”, is created with 4 replicas, all being a part of ”network1”.

Lastly, a new instance of type ”Node App” called ”other”, but without choosing a network, in other words, this node is not connected to any network.

More instances and networks could be added, however, it is not necessary for the purpose of testing. Having created the instances, the configuration can now be run, and no error should occur.



Name network1

Subnet 10.0.1.0/24

Gateway 10.0.1.1

Connect **network1** network to DNS? ✓

Network DNS Endpoint 10.0.1.2

Fig. 11. Setting up the ”Node App” called ”backend”

B. Testing

Testing starts by checking if the instances can communicate with each other, since they are in the same network, the result should be positive. Indeed, that is the obtained result, as seen in the UI (figure 12). By accessing a client instance container through the terminal, the ”backend” instance in the network ”network1” is pinged (10.0.1.3), and as shown in figure 13, they are indeed communicating.

Additionally, by accessing the instance called ”other”, the ”backend” instance is pinged and as it is shown in the figure 14, they are not communicating, as expected.

Next, we will test the DNS. A ping request will be made, just like before, but instead of pinging the ip address ”10.0.1.3” of the ”backend” instance, the address ”backend” is pinged, the DNS should then resolve the name.

The results are as shown in the figure 15. As expected and shown in the figure 16, the ”other” instance pinging the ”backend” instance using the DNS does not work, because ”other” is not registered in the DNS.

Lastly, the application shows statistics about the instances. To guarantee the correct functioning of these statistics, for example, if they show when a server is receiving many requests, a bash script with the following code `while true; do curl backend:5050; done` will be run in one of the client instances. It will be spamming requests to the ”backend” instance, this way, the CPU percentage on the ”backend” should rise along with some changes in some of the other indicators. The results are as shown in figures 17 and 18.

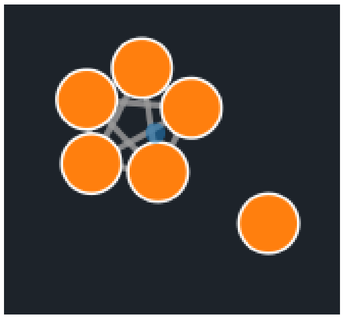
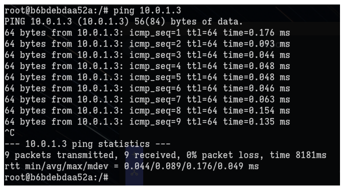
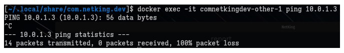


Fig. 12. Instances topology after setup



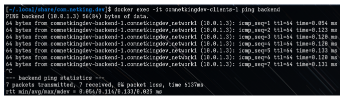
```
root@b6bdebdaa52a:/# ping 10.0.1.3
PING 10.0.1.3 (10.0.1.3) 56(84) bytes of data.
64 bytes from 10.0.1.3: icmp_seq=1 ttl=64 time=0.176 ms
64 bytes from 10.0.1.3: icmp_seq=2 ttl=64 time=0.093 ms
64 bytes from 10.0.1.3: icmp_seq=3 ttl=64 time=0.044 ms
64 bytes from 10.0.1.3: icmp_seq=4 ttl=64 time=0.048 ms
64 bytes from 10.0.1.3: icmp_seq=5 ttl=64 time=0.048 ms
64 bytes from 10.0.1.3: icmp_seq=6 ttl=64 time=0.046 ms
64 bytes from 10.0.1.3: icmp_seq=7 ttl=64 time=0.063 ms
64 bytes from 10.0.1.3: icmp_seq=8 ttl=64 time=0.154 ms
64 bytes from 10.0.1.3: icmp_seq=9 ttl=64 time=0.135 ms
^C
--- 10.0.1.3 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8181ms
rtt min/avg/max/mdev = 0.044/0.089/0.176/0.049 ms
root@b6bdebdaa52a:/#
```

Fig. 13. Pinging ”backend” instance from a client instance



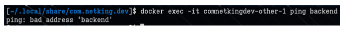
```
[~/.local/share/com.netking.dev]$ docker exec -it comnetkingdev-other-1 ping 10.0.1.3
PING 10.0.1.3 (10.0.1.3): 56 data bytes
^C
--- 10.0.1.3 ping statistics ---
14 packets transmitted, 0 packets received, 100% packet loss
```

Fig. 14. Failing to ping the ”backend” instance from ”other” instance



```
[~/.local/share/com.netking.dev]$ docker exec -it comnetkingdev-clients-1 ping backend
PING backend (10.0.1.3) 56(84) bytes of data.
64 bytes from comnetkingdev-backend-1.comnetkingdev_network1 (10.0.1.3): icmp_seq=1 ttl=64 time=0.054 ms
64 bytes from comnetkingdev-backend-1.comnetkingdev_network1 (10.0.1.3): icmp_seq=2 ttl=64 time=0.123 ms
64 bytes from comnetkingdev-backend-1.comnetkingdev_network1 (10.0.1.3): icmp_seq=3 ttl=64 time=0.120 ms
64 bytes from comnetkingdev-backend-1.comnetkingdev_network1 (10.0.1.3): icmp_seq=4 ttl=64 time=0.120 ms
64 bytes from comnetkingdev-backend-1.comnetkingdev_network1 (10.0.1.3): icmp_seq=5 ttl=64 time=0.133 ms
64 bytes from comnetkingdev-backend-1.comnetkingdev_network1 (10.0.1.3): icmp_seq=6 ttl=64 time=0.120 ms
64 bytes from comnetkingdev-backend-1.comnetkingdev_network1 (10.0.1.3): icmp_seq=7 ttl=64 time=0.131 ms
^C
--- backend ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6137ms
rtt min/avg/max/mdev = 0.054/0.114/0.133/0.025 ms
```

Fig. 15. Pinging ”backend” instance from a client instance using the DNS



```
[~/.local/share/com.netking.dev]$ docker exec -it comnetkingdev-other-1 ping backend
ping: bad address 'backend'
```

Fig. 16. Failing to ping the ”backend” instance from ”other” instance using the DNS

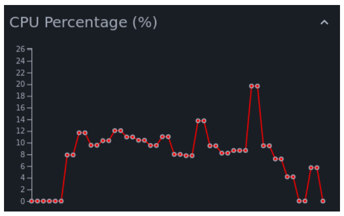


Fig. 17. CPU percentage of the ”backend” instance when receiving multiple requests

V. Conclusion

In this paper, we introduced a Network Management System that facilitates network visualization and virtualization. With

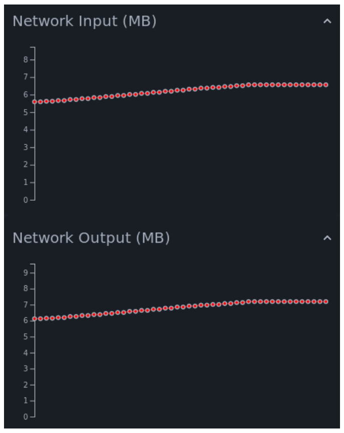


Fig. 18. Network Input and Output of the ”backend” instance when receiving multiple requests

the Tauri framework and the Rust programming language, we created an user-friendly cross-platform solution. This software makes it simple to create and manage instances, such as NodeApp, Client, Router, and Nginx. An extra element that made the project more educational was the incorporation of BIND for DNS management.

All factors considered, our Network Management System provides a complete answer for network administration needs, whilst being accessible to all users, even those not fluent in network administration.

References

[1] Faculdade de Engenharia da Universidade do Porto. URL: https://sigarra.up.pt/feup/en/UCURR_GERAL.FICHA_UC_VIEW?pv_ocorrencia_id=518819. (accessed: 30.05.2024).

[2] Rust. URL: https://www.rust-lang.org/. (accessed: 28.05.2024).

[3] SolarWinds. URL: https://www.solarwinds.com/network-management-software. (accessed: 27.05.2024).

[4] Tauri. URL: https://tauri.app/. (accessed: 28.05.2024).

[5] Wikipedia. URL: https://en.wikipedia.org/wiki/Tauri_(software_framework). (accessed: 28.05.2024).

[6] Wikipedia. URL: https://en.wikipedia.org/wiki/BIND. (accessed: 30.05.2024).