École Polytechnique Fédérale de Lausanne

Semester Project Report

Simple Network Emulator

Author
Ziyang Song (ziyang.song@epfl.ch)

Supervisor
Gauthier Voron (gauthier.voron@epfl.ch)

Laboratory
DCL - Distributed Computing Laboratory

School
IC - School of Computer and Communication Sciences

January 2024

Contents

1 Description

1.1 Basics

This project is forked from [1]. The original project is a simple network emulator that can be used to emulate a network topology and launch separate distributed terminals. It allows developers to assign a matrix of delays of the network and distinct psuedo IP addresses within a subnet for those distributed ends. It also provides various dummy examples to show case of the testing. It supports IPv4 and UDP. It is written in C++ and fit for plug-and-play. The original project is licensed under MIT License.

This project is now reposited at [2]. The goal of this project is to add more features to the original one and make it more practical. The features we added are listed below:

1. **Support for TCP**. We added support for TCP. We also added dummy examples with TCP recurring messages and file transfer.
2. **Test Running on BFT-SMaRt**. We tested running TCP with verifying the performance of BFT-SMaRt.
3. **Improvement on Configuration Parser and Debug Assistance**. We improved the configuration parser and added debug assistance.

1.2 Outlines

The project is organized as following figure 1 shows:

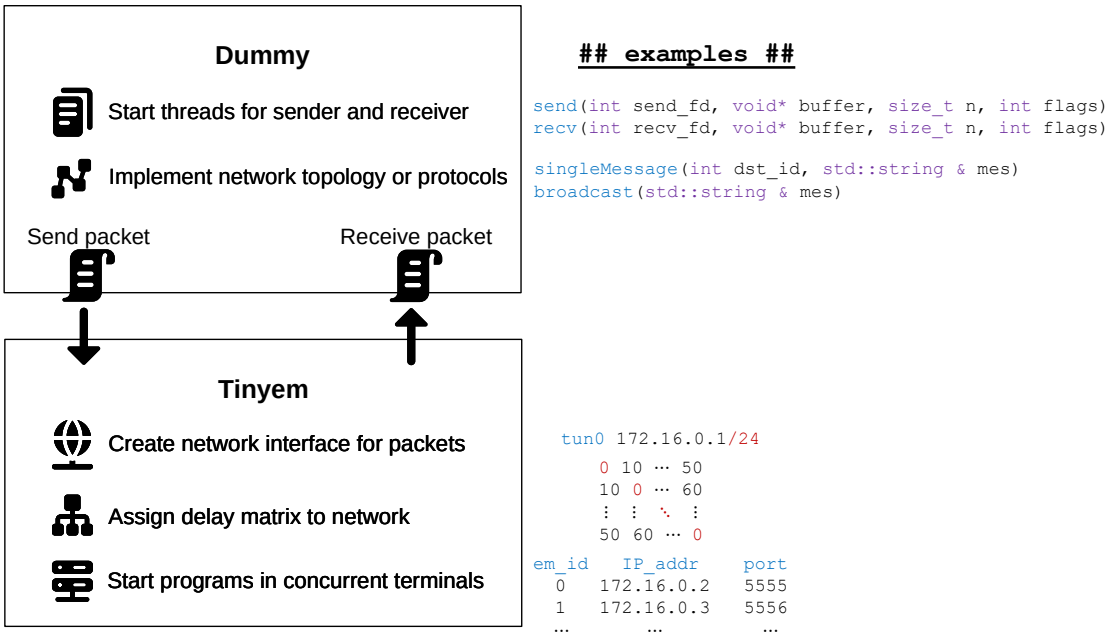


Figure 1: Outline of project

2 Installation

This installation guide is based on Ubuntu 18.04 (or higher version). It is assumed that the user has root access to the machine.

1. Clone the repository to your local directory on your machine, and `cd` to the root directory of the project.

```
cd /path/to/your/SimpleEM
```

2. Run cmake to generate the makefile. Please try clearing the `CMakeCache.txt` and `cmake_install.cmake` if error occured.

```
cmake .
```

3. Run make to compile the project. Please make sure that you successfully make the project by eliminating all the errors before running it.

```
make
```

4. Run the executable file `tinyem` to start the emulator. (notice that the default config file path is set to "`./configs/config.txt`", if you want to use other config file, please specify it as the first argument of the executable file)

```
sudo ./tinyem
```

   And you will see the following output on your terminal, which indicates the demo is running successfully:

```
[Server] 172.16.0.3, Listening on 5556
[Client] Socket Created!
[Client] Socket Connected!
[Client] Message Sent: Helloworld from client xx-xx-xx
[Server] 172.16.0.3:5556 Socket Accepted
[Server] Received: Helloworld from client xx-xx-xx
[Client] socket closed!
[Server] Children socket closed!
[Server] socket closed!
```

5. You can also run other executable files which the `CMakeLists.txt` specified.

   For example, `test_packet` demo allows you to input a hexstream which represents a TCP packet and test the result of the functions in `packet.hpp`;

```
./test_packet
## input the hex string e.g. 450000xxxx...
```

   And `dummy` demo allows you to test the TCP recurring message and file transfer locally (127.0.0.1), which would be introduced in section 4.

3 Machanism (`tinyem`)

3.1 Configuration Parser

The configuration parser is implemented in `config-parser.hpp`. Please read the appendix for the detailed documentation of the parser and how a config file is formed.

3.2 Network Emulator

Here we test the network emulator by taking the following default config file (`./config/config.txt`) as an example:

```
tun0 172.16.0.1 255.240.0.0
2
172.16.0.2 5555
172.16.0.3 5556
0 10
10 0
./tcp_client 172.16.0.3 5556
./tcp_server 172.16.0.3 5556
```

The `tinyem` turns on TUN interface and creates a virtual network interface `tun0` on the host machine. This interface is assigned with the IP address `172.16.0.1` with the subnet mask `255.240.0.0`.

It also creates two virtual network interfaces for two processes with their virtual IP address and virtual port respectively, with an assigned latency matrix. In this example, processes 0 (`172.16.0.2:5555`) and 1 (`172.16.0.3:5556`) are of a mutral latency of 10ms.

For process 0, it runs the executable file `./tcp_client` with the arguments `172.16.0.3 5556`, which represents the destination IP address and destination port of the server.

For process 1, it runs the executable file `./tcp_server` with the arguments `172.16.0.3 5556`, which represents the listening IP address and listening port of the server.

The figure 2 shows the TCP packets during the execution of the demo:

No.	Source	Destination	Protocol	Length	Info
51	172.16.0.1	172.16.0.3	TCP	76	57838 → 5556 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=1101353423 TSecr=0 WS=128
52	172.16.0.2	172.16.0.1	TCP	76	57838 → 5556 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=1101353423 TSecr=0 WS=128
53	172.16.0.1	172.16.0.2	TCP	76	5556 → 57838 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=2796299953 TSecr=1101353423 WS=2
54	172.16.0.3	172.16.0.1	TCP	76	5556 → 57838 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=2796299953 TSecr=1101353423 WS=2
55	172.16.0.1	172.16.0.3	TCP	68	57838 → 5556 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=1101353463 TSecr=2796299953
56	172.16.0.1	172.16.0.3	TCP	99	57838 → 5556 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=31 TSval=1101353463 TSecr=2796299953
57	172.16.0.2	172.16.0.1	TCP	68	57838 → 5556 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=1101353463 TSecr=2796299953
58	172.16.0.2	172.16.0.1	TCP	99	57838 → 5556 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=31 TSval=1101353463 TSecr=2796299953
59	172.16.0.1	172.16.0.2	TCP	68	5556 → 57838 [ACK] Seq=1 Ack=32 Win=64210 Len=0 TSval=2796300001 TSecr=1101353463
60	172.16.0.3	172.16.0.1	TCP	68	5556 → 57838 [ACK] Seq=1 Ack=32 Win=64210 Len=0 TSval=2796300001 TSecr=1101353463
61	172.16.0.1	172.16.0.3	TCP	68	57838 → 5556 [FIN, ACK] Seq=32 Ack=1 Win=64256 Len=0 TSval=1101354471 TSecr=2796300001
62	172.16.0.2	172.16.0.1	TCP	68	57838 → 5556 [FIN, ACK] Seq=32 Ack=1 Win=64256 Len=0 TSval=1101354471 TSecr=2796300001
63	172.16.0.1	172.16.0.2	TCP	68	5556 → 57838 [FIN, ACK] Seq=1 Ack=32 Win=64210 Len=0 TSval=2796301009 TSecr=1101353463
64	172.16.0.1	172.16.0.2	TCP	68	5556 → 57838 [ACK] Seq=2 Ack=33 Win=64210 Len=0 TSval=2796301009 TSecr=1101354471

Figure 2: Packet Captrued by Wireshark

As we can see, there are two TCP connections established between the client and the server. The first one (`172.16.0.1:57838 -> 172.16.0.3:5556`) is that TUN acts as process

0 and simulates sending the packet to process 1. The second one (`172.16.0.2:57838 -> 172.16.0.1:5556`) is that TUN acts as process 1 and simulates receiving packet from process 0.

The TUN actually acts as a gateway, modifys the TCP packets' headers and sends them to the virtual destination or receives them from the virtual source, while adding the latency to the packets.

Notice the host IP addresses of process 0, or process 1 are not real existing, but TUN runs with time division multiplexing, which means that it acts as process 0 for a while and process 1 for a while. In this case, a single TUN interface can simulate multiple processes in a distributed system.

Also notice that the `tinyem` works for assigning IP addresses and ports for the processes and handling the latency between these processes, and it behaves opaque to the upper layer application. In other word, the parsed process does not need to know the existence of the TUN interface, nor to rely on any downer layer functions, and it can run locally without this emulator as if it is running on a real distributed system.

For detailed information, please read the appendix for the documentation of orginal project report of the network emulator.

4 TCP Demo (dummy)

The TCP demo is implemented in `dummy.cpp`, providing some simple use cases of algorithms to test the emulator. The demo is class-based, where the inheritation relationship could be found in appendix file. The input arguments are `em_id` and `configPath`, for example:

```
dummy 0 ./examples/dummy/config/config.txt
```

where the config path points to the config file containing a list of IP addresses and ports of the processes, which would be parsed over the initialization of `network-helper.hpp`. Notice that the `em_id` is preset to unsigned integer index in order of the config file (starting from 0).

4.1 Packet Recursive Transmission

This demo is implemented in `tcp-peer.hpp`. It is a simple recursive transmission of a TCP packet. The packet is sent from a peer process to another peer process (a peer process each contains two threads, one for sending and one for receiving, making it a duplex-channel.), and it is sent back reversively. Each peer could also be understood as a simple echo server, doing so-called "ping-pong" transmission.

Note: You need to revise the `tcp-peer.hpp` by uncommenting the code for packet recursive transmission (follow comment indication "`For ping-pong packet test`"), and compile it. Then run the `tinyem` demo with configuration file `./configs/config5.txt`, which would generate 5 processes and process 0 is initialized to start sending "ping" to the other processes, and other process would echo "pong" to it and go on. You will see the output like the following on your terminal:

```
[dummy] 4/5
[network-helper] 4 Listening on 172.16.0.6:5559
...
[tcp-peer] send_thread sending:0->4 ping
[network-helper] Src: 0, Des: 4 172.16.0.6:5559
[network-helper] 4 Socket on server, Accepted
[tcp-peer] 4 GOT FROM 0 MESSAGE: ping
...
[tcp-peer] send_thread sending:4->0 pong
[network-helper] Src: 4, Des: 0 172.16.0.2:5555
[network-helper] 0 Socket on server, Accepted
[tcp-peer] 0 GOT FROM 4 MESSAGE: pong
...
[network-helper] 0 Listening on 172.16.0.2:5555
[tcp-peer] send_thread waiting...
...
```

Due to the property of asynchronous system, the output of the demo is not in the same order. However, the demo is still working as expected.

Noticablly, the function `receive_tcp()` in `tcp-peer.hpp` is designed to tell the sender's process id trivially by matching the source IP address of the packet with the entry of the vector `addresses`. It is not a good design, since the IP address is not a unique identifier of a process

and it would be conflicted and buggy if you try to run it locally with same IP addresses (127.0.0.1). You can modify the function or find some encoding to make it work with other unique identifiers.

4.2 File Transfer

This demo is implemented a simple file transfer from a peer process to another peer process. The file is loaded, get the size of it, divided into smaller chunks and sent to the other peer process in order. The other peer process would receive the chunks and write them into a file until meeting the size of the file if the transfer is successful.

This demo takes most trivial fail-stop mechanism, with simply a long enough timegap for the network and receiver to congest a chunk, without considering order encoding, socket blocking, interupt and recover, etc. Therefore, it couldn't work well (error occurs) if there is more than 1 chunk, (within the chunk, TCP is reliable for the completeness and ordering throughout the transmissions, even there are multiple packets) and is not a good design, but it is enough to show the basic idea of file transfer.

You need to revise the `tcp-peer.hpp` by uncommenting the code for file transfer (follow comment indication "`For large file transfer test`"), changing the file with the corresponding filepath you want to transfer, and compile it. Then run the `tinyem` demo with configuration file `./configs/config2.txt`, which would generate 2 processes and process 0 is initialized to start sending the file to process 1. You will see the output like the following on your terminal and you may see the copyed and renamed file in the directory of the receiver:

```
[dummy] 0/2
[tcp-peer] send_thread sending:0->1 epfl-logo.svg
[network-helper] Src: 0, Des: 1 172.16.0.3:5556
[network-helper] 0 Listening on 172.16.0.2:5555
[dummy] 1/2
[network-helper] 1 Listening on 172.16.0.3:5556
[network-helper] sender: socket connected
[sendFile] Filesize: 1254
[network-helper] 1 Socket on server, Accepted
[recvFile] Filesize: 1254
[sendFile] File: epfl-logo.svg, Filesize: 1254
[network-helper] sent: 1254/1254
[network-helper] recv: 1254/1254
[tcp-peer] 1 GOT FROM 0 MESSAGE: xx-xx-xx-xxxx.svg
[network-helper] 1 Listening on 172.16.0.3:5556
[tcp-peer] send_thread waiting...
...
```

4.3 Other Demos

There are other demos under the `dummy` directory, mostly inherited from the previous project and functions with class `AlgorithmBase`, which could be examples to adapt to various distributed scenarios. You can also write your own demo and run it with the emulator.

5 BFT-SMaRt Test

Byzantine Fault-Tolerant (BFT) State Machine Replication (SMaRt) [3], is a high-performance Byzantine fault-tolerant state machine replication project named BFT-SMaRt, a Java open source library maintained by the LaSIGE research unit at the University of Lisbon.

The BFT-SMaRt project uses TCP packets to communicate between the replicas, hence the primitive implementation of a TCP network emulator would be a good and practical premise for running testing.

Here we tested the BFT-SMaRt project with the emulator by the following step:

1. Download the BFT-SMaRt project, correctly configure the environment and parameters. Here we take example of its counter demo and fill in the `config/hosts.config` file with the IP addresses and ports of the simulated processes as follows:

```
# For Server
0 172.16.0.1 10000 10001
1 172.16.0.1 10010 10011
2 172.16.0.1 10020 10021
3 172.16.0.1 10030 10031
# For Client
1001 172.16.1.6 10100
```

   Here the configuration is different from that of emulator, because for the server in the BFT-SMaRt project, it needs a real IP address be assigned to bind to and listen on, but from the emulator's point of view, the servers are just processes with the given virtual IP addresses and virtual ports.

2. Then, compile it and get the compiled jar files and find following folder structure:

```
/path/to/your/BFT-SMaRt-library
|
+---build
|   |
|   +---install
|   |   |
|   |   \---YourProjectName
|   |       |   smartrun.sh
|   |       |   ...
...
```

3. You need to manually `cd` to the directory where `smartrun.sh` is located, and run the following command to start the BFT-SMaRt demo project with the emulator:

```
sudo /path/to/your/SimpleEM/tinyem /path/to/your/SimpleEM/configs/
config_bftsmart.txt
```

   It would start all the 4 servers with the emulator, but without client. Because the server needs time to setup and be ready. You can see the output like the following on your terminal:

```
 -- Using view stored on disk
 -- SSL/TLS handshake complete!, Id:0  ## CipherSuite:
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256.
 -- ID = 0
 -- N = 4
 -- F = 1
 -- Port (client <-> server) = 10000
 -- Port (server <-> server) = 10001
 -- requestTimeout = 2000
 -- maxBatch = 1024
 -- Using Signatures
 -- Binded replica to IP address 172.16.0.1
 -- SSL/TLS enabled, protocol version: TLSv1.2
 -- In current view: ID:0; F:1; Processes:0(/172.16.0.1:10000)
,1(/172.16.0.1:10010),2(/172.16.0.1:10020),3(/172.16.0.1:10030),
 -- Replica state is up to date
 --
                ###################################
                    Ready to process operations
                ###################################
```

4. Then, you can start the client in a new terminal, `cd` to the same directory and with the following command:

```
./smartrun.sh bftsmart.demo.counter.CounterClient 1001 1 10
```

   And you can see the output like the following on your server's terminal, showing the counter demo in successfully running with the emulator:

```
(1) Counter was incremented. Current value = 1
...
(10) Counter was incremented. Current value = 10
```

5. Note that in the `config_bftsmart.txt` we use `java` as executable which is copied directly from `smartrun.txt`, because `tinyem` couldn't take whole shell script as an executable program. Now you can test performance with other class in BFT-SMaRt project, or even your own project.

Since the project is written in Java, we can also use Java Native Interface (JNI) [4] to call the Java functions in C++, which would be efficient in future development. The JNI is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly.

6 Future Improvement

There are a lot of aspects to be improved and tested in the project. Here we list some of them:

6.1 Emulator Part (`tinyem`)

1. Use IP header's protocol field to distinguish TCP and UDP packets and handle them differently.
2. Assure the correctness for checksum of the packets.
3. Adding support for IPv6.
4. Design a GUI for this project to increase its readability.
5. ...

6.2 Demo Part (`dummy`)

1. Implemente a more practical and robust file transfer demo and case dealing with error.
2. Implemente more distributed algorithms with a better encoding data structure and run simulations on them.
3. ...

6.3 BFT-SMaRt Part

1. Explain the behavior of the BFT-SMaRt project with the emulator with a more complicated test and a more quantified experimental method.
2. Try to use JNI to call the Java functions in C++ to handle with the executable program problem and the asynchrony between server and client, improving the efficiency.
3. ...

References

[1] The original project repository, https://github.com/MirazSpecial/SimpleEM

[2] The current project repository, https://github.com/Chiron19/SimpleEM

[3] BFT-SMaRt, https://bft-smart.github.io/library/

[4] JNI, https://docs.oracle.com/javase/7/docs/technotes/guides/jni

SimpleEM

Semester project report

Konrad Litwiński

June 2023

Contents

1 Abstract

Testing plays a vital role in the development of reliable and robust applications in the field of computer science. The process of testing allows for empirical verification of application correctness and evaluation of performance across diverse scenarios. However, when it comes to applications intended to run in distributed environments, testing poses unique challenges. Cloud testing, while providing realism, can be prohibitively expensive, even for simple applications. On the other hand, local testing restricts application resources by forcing coexistence with other instances on a single machine.

In this project, we introduced SimpleEM, a novel approach that combines the realism of cloud testing with the cost-effectiveness of local testing by emulating a distributed environment on a single machine. By leveraging advanced emulation techniques, SimpleEM overcomes the limitations of existing testing methods. It creates a simulated distributed environment within a local machine, utilizing process scheduling and network simulation technologies.

2 Introduction

There are some already existing network emulators which could help in distributed application testing - ns3 or PeerSim to name a few. To describe exactly what those emulators lack, and what SimpleEM provides, lets specify four main characteristics that we would like our system to have:

2.1 Locality

As mentioned earlier, SimpleEM aims to achieve cost-effectiveness by running both the system and the application it tests on a single machine. However, this approach presents two significant challenges that SimpleEM effectively addresses.

The first challenge is the limitation of application resources. When multiple instances of the application run on a single machine, they must share resources that would otherwise be independent if they were executed on separate machines. SimpleEM tackles this issue by efficiently managing and allocating resources, ensuring that each instance receives the necessary resources for comprehensive testing.

The second challenge is related to communication between nodes in a distributed environment. In a real-world distributed setup, issues like packet dropping and substantial latency can occur, affecting the reliability and performance of the system. However, when running on a single machine, the communication between nodes lacks the real-world challenges, making the testing process less realistic.

2.2 Transparency

This characteristic is described as the first challenge in **Locality** point. It states, that from the point of view of the application it has all the resources available to itself as if it was running alone on a single machine. This in particular means that it cannot run in parallel with another instance of the application, or that the SimpleEM system cannot add a substantial computational overhead. Neither ns3 nor PeerSim do not fulfill this requirement.

2.3 Realism

This characteristic is described as the second challenge in **Locality** point. It refers to realism in communication, so in packet dropping and latency. In particular, through the testing process the developer should be able to test the application performance for different scenarios of pairwise latencies between its instances. What that means is the SimpleEM system should have some sort of control over the packet delivery process. This level of control enhances the testing process, allowing for comprehensive evaluation of the application's behavior under diverse communication conditions.

2.4 Plug-and-play

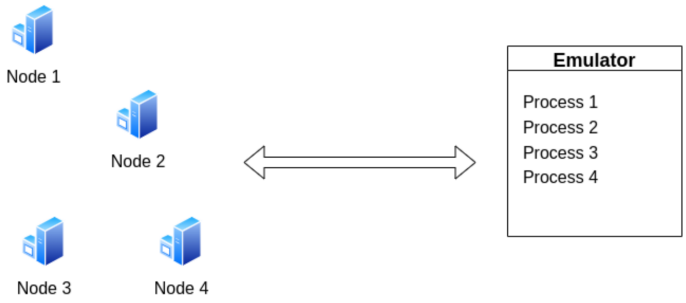
Forcing the developer to introduce some changes to the application code to test it causes many problems. The most obvious of them is that this extends the testing process. Other big problem is potential logic changes introduced while rewriting code for testing. This characteristic states that testing should be possible when the application code is not available. And here again, neither ns3 nor PeerSim do not fulfill this requirement.

3 Emulator overview

SimpleEM emulates the distributed environment ensuring the above mentioned four characteristics. Emulator's logic can be broken down into three following main points

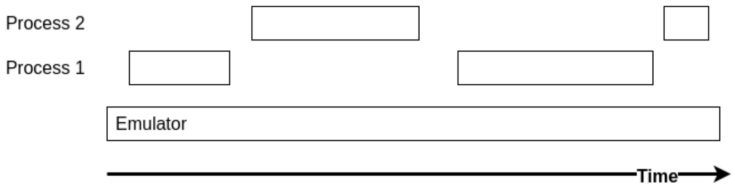
3.1 Processes as nodes

Emulator creates as many standard Unix processes as there are nodes in an emulated system. The application is run independently on every process. This ensures **Locality** and **Plug-and-play**.



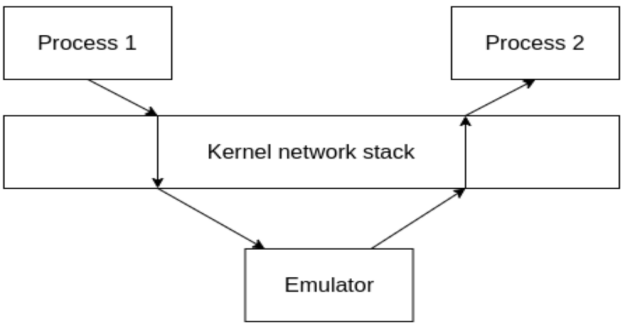
3.2 Process scheduling

To ensure **Transparency** we make sure that at any point in time at most one of the emulated processes is running. In particular, SimpleEM decides which process to 'schedule' - that is, which process to run and for how long. This means that for any given process, all the system resources are available at any point in time at which it's running.



3.3 Packet delivery control

Even though, from the point of view of application packets are send and received normally, SimpleEM decides at what point in time of the process execution the packet should be delivered. This allows us to ensure **Realism**.



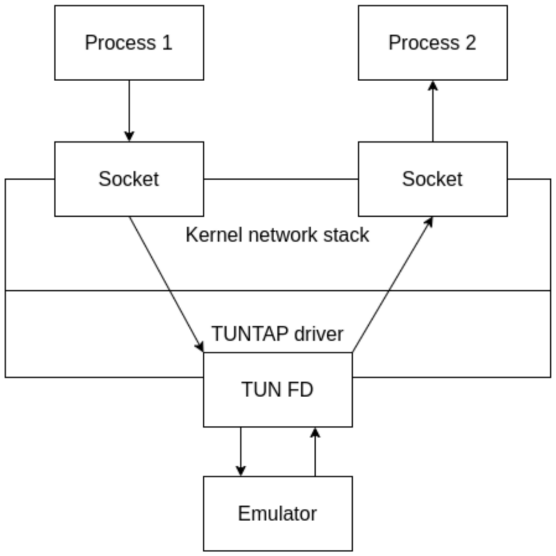
4 Emulator implementation

The detailed documentation of SimpleEM code can be found at the end of this PDF. Here we just give an intuition of the main elements of the emulator.

4.1 Packet control

SimpleEM incorporates packet control functionality by intercepting packets exchanged between different processes in the system using the TUN/TAP interface, specifically utilizing the TUN version. The TUN and TAP interfaces are virtual network devices implemented entirely in software, eliminating the need for physical network adapters. When a user sends a packet to an address in the TUN subnetwork, it traverses the network stack in the kernel. However, instead of being transmitted through a physical network adapter, the packet's IP frame (or Ethernet frame in the case of TAP) is written to a designated TUN file descriptor (FD).

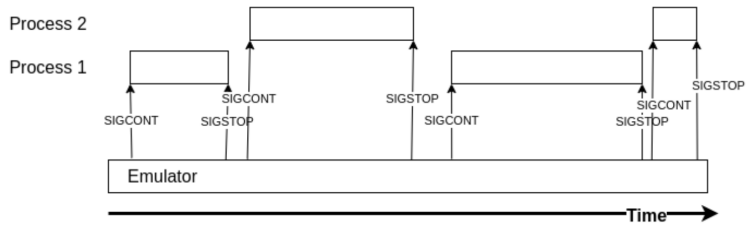
Similarly, packets sent using the TUN FD are written to the TUN interface, where they undergo processing within the kernel's network stack, just like any other received IP frame. Upon intercepting a transmitted packet, SimpleEM takes control of deciding when, or if, to deliver it to the intended destination process. This interception mechanism empowers SimpleEM to introduce packet dropping or latency, enabling developers to simulate real-world network conditions during the testing process.



4.2 Process control

To control the execution of processes, SimpleEM leverages the POSIX SIGSTOP and SIGCONT signals. Once SimpleEM creates emulated processes, it promptly stops their execution by sending the SIGSTOP signal. To resume a process for a specific duration, SimpleEM utilizes the SIGCONT signal, temporarily "awakening" the process. Subsequently, after a specified amount of time, SimpleEM halts the process again using the SIGSTOP signal. Importantly, these signals cannot be masked or ignored, ensuring that process control remains effective regardless of the application's functionality.

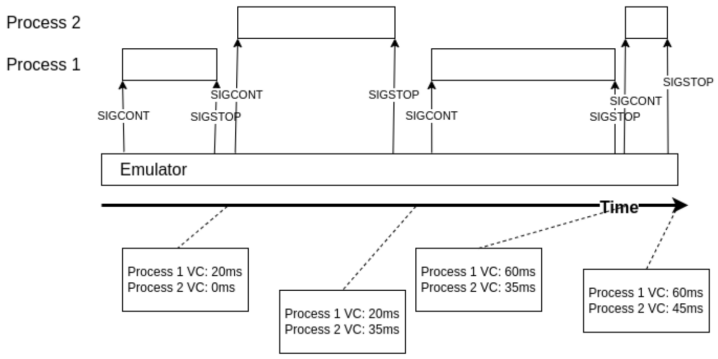
To synchronize with the process state changes, SimpleEM waits for the SIGCHLD signal, which indicates that a process has indeed transitioned to a new state. This ensures that there are no race conditions where two processes might be running simultaneously. By coordinating with the SIGCHLD signal, SimpleEM guarantees accurate process management and avoids any conflicts that could arise from concurrent execution.



4.3 Time control

To accurately measure the duration of each process's execution, SimpleEM introduces the concept of a virtual clock. For every emulated process, SimpleEM maintains a virtual clock that increments when the process is actively executing. Initially, all processes start at the same point in time, with their virtual clocks set to zero. As a process runs, its virtual clock increases, reflecting the amount of time it has traversed within the emulated scenario.

It is crucial to note that during the emulator's execution, processes will have different values on their respective virtual clocks. This disparity arises due to variations in execution times and interactions with the emulated environment. By tracking individual virtual clocks, SimpleEM provides a comprehensive understanding of the progress and timing of each process within the emulated scenario.



4.4 Process awakening

Let's consider the following scenario: we have two processes, p_1 with virtual clock t_1 and p_2 with virtual clock t_2, and their pairwise latency is denoted as δ. In this scenario, if at any point during the emulation we observe that $t_1 + \delta < t_2$, it raises concerns about correctness.

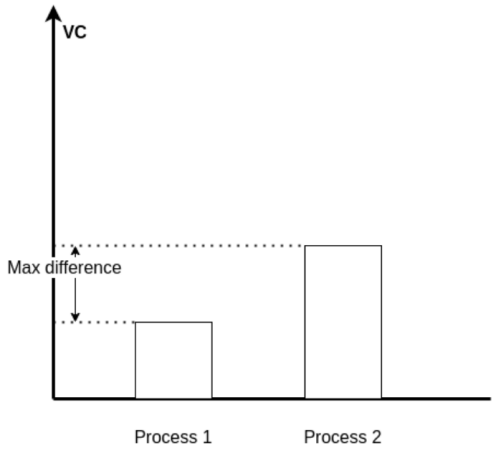
To illustrate this, suppose p_1 sends a packet at time t_1, expecting it to be delivered to p_2 at time $t_1 + \delta$. However, if p_2 has already progressed beyond $t_1 + \delta$, it implies that its behavior was not influenced by the reception of this packet, despite the fact that it could have been in reality. This discrepancy highlights the importance of maintaining synchronization between processes.

In a system comprising n processes denoted as $p_1, p_2, \ldots, p_n$, each with its corresponding virtual clock $t_1, t_2, \ldots, t_n$, and pairwise latencies represented by δ_{ij}, it is crucial to ensure that the following condition holds true at all times:

$$\forall_{i,j \in \{1 \ldots n\}} |t_i - t_j| \leq \delta_{ij}$$

Having that in mind, the way that SimpleEM schedules processes is by repeating the following

- Choosing process p_i with the lowest value of virtual clock
- Awakening p_i for the longest time which would not violate the above mentioned condition

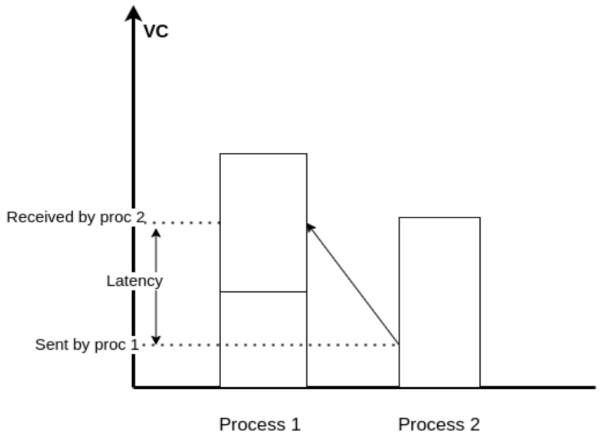


4.5 Latency control

Finally, SimpleEM provides the capability to introduce user-defined latencies between processes by strategically determining when to deliver packets. When a process p_i is awakened, all packets sent by that process are stored in the corresponding *out_packets* queue. Once p_i is stopped, SimpleEM iterates through the *out_packets* queue, moving the packets to the appropriate *in_packets* queues of the destination processes.

For a packet sent from p_i to p_j at time t_i, SimpleEM sets the expected delivery time of the packet to $t_i + \delta_{ij}$. Subsequently, when p_j is awakened and its virtual clock reaches $t_i + \delta_{ij}$, SimpleEM delivers the packet to p_j.

This mechanism allows SimpleEM to introduce the desired latencies between processes, ensuring that packets are delivered at the appropriate times according to the specified pairwise latencies. By controlling the delivery timing in this manner, SimpleEM enables developers to accurately simulate communication delays and assess the impact of latency on the behavior and performance of the emulated distributed system.


VC
Received by proc 2
Latency
Sent by proc 1
Process 1
Process 2

5 Conclusion

In conclusion, SimpleEM provides a powerful solution for testing distributed applications by combining the realism of cloud testing with the cost-effectiveness of local testing. By emulating a distributed environment on a single machine, SimpleEM overcomes the limitations of resource sharing and communication challenges encountered in traditional local testing setups.

SimpleEM incorporates various key features to enhance the testing process. The packet control mechanism allows developers to introduce controlled packet dropping and latency, enabling the simulation of real-world network conditions. Additionally, the utilization of POSIX signals, such as SIGSTOP and SIGCONT, facilitates efficient process control and synchronization, ensuring accurate emulation of process behavior.

The virtual clock mechanism in SimpleEM accurately measures the duration of each process's execution, enabling precise tracking of progress within the emulated scenario. This feature provides developers with valuable insights into the timing and synchronization of processes, contributing to thorough testing and analysis of distributed applications.

Furthermore, SimpleEM allows for the introduction of user-defined latencies, enabling the testing of application behavior under various communication delay scenarios. By controlling the delivery timing of packets, developers can assess the impact of latency on system performance and ensure the correctness of inter-process communication.

Overall, SimpleEM offers a robust and cost-effective solution for testing distributed applications. Its ability to emulate a distributed environment on a single machine, coupled with its advanced features, empowers developers to conduct reliable, comprehensive, and realistic testing, leading to the development of more resilient and efficient distributed systems.

SimpleEM

Generated by Doxygen 1.9.1

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

Class Documentation

2.1 ConfigParser Class Reference

Class parsing and saving the configuration from a file.

`#include <config-parser.hpp>`

Public Member Functions

- ConfigParser (const std::string &config_path)

  Reads config from the config file (`config_path`)

Public Attributes

- std::string tun_dev_name

  Device name for new TUN interface.
- std::string tun_addr

  Address of the TUN interface.
- std::string tun_mask

  Mask of the TUN interface subnetwork.
- int procs

  Number of processes to be emulated by the emulator.
- std::vector< std::vector< int > > latency

  Matrix of pairwise latencies.
- std::vector< std::pair< std::string, int > > addresses

  Addresses (in number/dot format) and ports of processes.
- std::vector< std::string > program_paths

  Paths to programs to be run on every process.
- std::vector< std::string > program_names

  Names of programs to be run on every process.
- std::vector< std::vector< std::string > > program_args

  Arguments to be passed to every process.

2.1.1 Detailed Description

Class parsing and saving the configuration from a file.

2.1.2 Constructor & Destructor Documentation

2.1.2.1 ConfigParser()

```
ConfigParser::ConfigParser (
            const std::string & config_path )
```

Reads config from the config file (`config_path`)

Config layout is as follows:

- First line consists of three words split by whitespace, tun device name, tun interface address, and tun interface address mask, to be used in setting up the tun interface.
- Second line consists of one number - procs - num of procs to simulate
- Next procs lines consists of a string and int each, i-th line means the address and port on which i-th proc is listening
- Next procs lines consists of procs numbers each, i-th number in j-th line means the latency from i-th to j-th proc in miliseconds (i-th number in i-th column should be 0)
- Next procs lines consist of at least two words each, i-th line starts with the path to i-th program, then the name of the i-th program and then whitespace-split args to the i-th program.

The documentation for this class was generated from the following file:

- src/include/config-parser.hpp

2.2 EMProc Class Reference

Controller of a single process inside an emulation.

`#include <emproc.hpp>`

Public Member Functions

- EMProc (em_id_t em_id, int pid)
  Class main constructor.
- void awake (struct timespec ts, const Network &network)
  Awake emulated process and let him run for specified amount of time, intercepting packets sent by it.

Public Attributes

- em_id_t em_id

  Internal id of emulated process.
- int pid

  pid of emulated process
- struct timespec virtual_clock

  Time that the process was awake.
- std::priority_queue< Packet > out_packets

  Buffer of packets sent by process.
- std::priority_queue< Packet > in_packets

  Buffer of packets to be received by process.

Private Member Functions

- bool to_receive_before (struct timespec ts)

  Check if there is any packet that this process should receive before the specified timestamp.

2.2.1 Detailed Description

Controller of a single process inside an emulation.

Class responsible for the state and awakening of a single process in an emulation. It holds the queues of packets that were sent from the process as well as the queue of packets that should be delivered to the process. It keeps track of process virtual clock. All the inside-awakening logic is kept in this class.

2.2.2 Constructor & Destructor Documentation

2.2.2.1 EMProc()

```
EMProc::EMProc (
            em_id_t em_id,
            int pid )  [inline]
```

Class main constructor.

Sets process' virtual clock to 0.

Parameters

em_id	Emulator's internal process id of this process
pid	Operating systems pid of process associated with this emulated process

2.2.3 Member Function Documentation

2.2.3.1 awake()

```
void EMProc::awake (
            struct timespec ts,
            const Network & network )
```

Awake emulated process and let him run for specified amount of time, intercepting packets sent by it.

Awakes emulated process for amount of time specified by `ts` . All packets sent by this process to addresses in the subnetwork of TUN interface specified by `tun_fd` will be intercepted and stored in out_packets.

Function sends SIGCONT signal to specified process, then after `ts` time it sends the SIGSTOP signal. During the time that the process is running all packets sent by it will be intercepted. Also, in appropriate times, packets from in_packets will be sent to the process using the TUN interface with `tun_fd` . virtual_clock is updated inside this function. It is guaranteed that when the function returns, the specified process has already stopped.

Parameters

ts	Time for the process to run
network	Network on which the simulator is operating

2.2.3.2 to_receive_before()

```
bool EMProc::to_receive_before (
            struct timespec ts )  [private]
```

Check if there is any packet that this process should receive before the specified timestamp.

Checks if the first packet in in_packets buffer exists and has timestamp lower then the one specified by `ts`

Parameters

ts	Time for which to check

Returns

If such packet exists

The documentation for this class was generated from the following file:

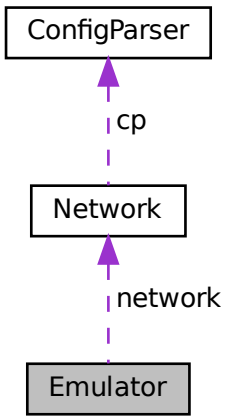
- src/include/proc_control/emproc.hpp

2.3 Emulator Class Reference

Class encapsulating the main logic of the emulator.

`#include <emulator.hpp>`

Collaboration diagram for Emulator:



Public Member Functions

- Emulator (Network &network, const ConfigParser &cp)

  Creates an emulation and spawns specified number of new processes.
- void start_emulation (int steps)

  Starts the emulation by iteratively scheduling processes.
- void kill_emulation ()

  Kills all spawned processes.

Private Member Functions

- void fork_stop_run (int *pids, const ConfigParser &cp)

  Forks the process and initializes child processes.
- void child_init (const std::string &program_path, const std::string &program_name, const std::vector< std::string > &program_args)

  Initialize the child process.
- em_id_t choose_next_proc () const

  Chooses next process to be scheduled for awakening.
- struct timespec get_time_interval (em_id_t em_id) const

  Returns the longest time `em_id` can run without violating correctness.
- void schedule_sent_packets (em_id_t em_id)

  Moves packets sent by `em_id` to appropriate in queues of receiving processes.

Private Attributes

- int procs

  Number of processes being emulated.
- std::vector< EMProc > emprocs

  States of each process.
- Network & network

  Specifies network on which the emulation is being run.

2.3.1 Detailed Description

Class encapsulating the main logic of the emulator.

Class responsible for overall control of the emulation. It creates specified number of new processes and runs the emulated program on them. It controls the state of those processes (whether they are stopped or running), and decides which one should be awaken (scheduled) next and for how lond. At last, it is responsible for the cleanup after the emulation is finished.

2.3.2 Constructor & Destructor Documentation

2.3.2.1 Emulator()

```
Emulator::Emulator (
            Network & network,
            const ConfigParser & cp )
```

Creates an emulation and spawns specified number of new processes.

Creates the emulation by forking the process procs times. Every newly created process immediately stops itself by the
`raise(SIGSTOP)`

call. Objects corresponding to newly created processes are stored in the emprocs vector.

Parameters

network	The network on which the emulator runs
cp	Configuration of the emulation

2.3.3 Member Function Documentation

2.3.3.1 child_init()

```
void Emulator::child_init (
            const std::string & program_path,
```

```
            const std::string & program_name,
            const std::vector< std::string > & program_args )  [private]
```

Initialize the child process.

Start by doing the
`raise(SIGSTOP)`

call. After the process is awaken (using the
`SIGCONT`

signal), it executes the program specified by `program_path`.

Parameters

program_path	Path to the program to be executed on this process
program_name	Name of the program to be executed on this process
program_args	Arguments to be passes to the program

2.3.3.2 choose_next_proc()

```
em_id_t Emulator::choose_next_proc ( ) const  [private]
```

Chooses next process to be scheduled for awakening.

Loops through all te processes and choses the one which was executed for the least amount of time.

Returns

Id of the process to be scheduled next

2.3.3.3 fork_stop_run()

```
void Emulator::fork_stop_run (
            int * pids,
            const ConfigParser & cp )  [private]
```

Forks the process and initializes child processes.

Creates procs new processes that execute the child_init function. Saves process ids of newly created processes in the `pids` array.

Parameters

pids	Array in which to store process ids of new processes
cp	Configuration to be used for new processes initialization

2.3.3.4 get_time_interval()

```
struct timespec Emulator::get_time_interval (
            em_id_t em_id ) const  [private]
```

Returns the longest time `em_id` can run without violating correctness.

Every process has its virtual clock saved, returns minimum (over all processes p != `em_id`) of p->virtual_clock - em_id ->virtual_clock + network.get_latency(p, em_id)

Parameters

em↩ _id	Process which will be run

Returns

The maximum possible time to run

2.3.3.5 kill_emulation()

```
void Emulator::kill_emulation ( )
```

Kills all spawned processes.

Kills all spawned processes, effectively ending the emulation.

2.3.3.6 schedule_sent_packets()

```
void Emulator::schedule_sent_packets (
            em_id_t em_id )  [private]
```

Moves packets sent by `em_id` to appropriate in queues of receiving processes.

For every packet sent by the `em_id`, this function moves it to a queue of packets awaiting to be received by appropriate other process. The function increases the timestamp of the packet by the pairwise latency between those two processes - so that the packet will be received at proper time.

Parameters

em↩ _id	Id of the process whose out packets need to be moved

2.3.3.7 start_emulation()

```
void Emulator::start_emulation (
            int steps )
```

Starts the emulation by iteratively scheduling processes.

Performs `steps` times the loop of choosing the next process to schedule, calculating for what time it should run, awakening it for that time, and later sorting packets sent by it to appropriate queues.

Parameters

steps	Number of awakenings to perform

The documentation for this class was generated from the following file:

- src/include/emulator.hpp

2.4 Logger Class Reference

Utility class for logging in the system.

`#include <logger.hpp>`

Public Member Functions

- Logger (const std::string &file_path)
  Initializes logger and opens specified file.
- void log_event (clockid_t clock_type, const char *format, va_list args)
  Log specified formatted string (printf style) with time of given clock.
- void log_event (clockid_t clock_type, const char *format,...)
  Log specified formatted string (printf style) with time of given clock.
- void log_event (const char *format,...)
  Log specified formatted string (printf style) with CLOCK_MONOTONIC.

Static Public Member Functions

- static void dump (const char *buf, size_t len)
  Print given buffer to stdout in hex and bin.
- static size_t push_to_buffer_time_safe (char *buf, clockid_t clk_id)
  Appends current time to buffer (async-signal-safe)
- static size_t push_to_buffer_string_safe (char *buf, const char *expr)
  Appends string to buffer (async-signal-safe)
- static size_t push_to_buffer_int_safe (char *buf, int expr)
  Appends int to buffer (async-signal-safe)
- static void print_string_safe (const std::string &expr)
  Prints string to STDOUT (async-signal-safe)
- static void print_int_safe (int expr)
  Prints int to STDOUT (async-signal-safe)
- static void print_time_safe (clockid_t clk_id)
  Prints time to STDOUT (async-signal-safe)

Private Member Functions

- void log_time (clockid_t clock_type)

  Appends time to logging file (without adding endline after)

Private Attributes

- FILE * file_ptr

  Pointer to FILE object to which to write.

2.4.1 Detailed Description

Utility class for logging in the system.

Logger is used by the emulator for logging all recorded events in a structured manner.

2.4.2 Constructor & Destructor Documentation

2.4.2.1 Logger()

```
Logger::Logger (
            const std::string & file_path )
```

Initializes logger and opens specified file.

Parameters

file_path	Path to file to which to log to

2.4.3 Member Function Documentation

2.4.3.1 dump()

```
void Logger::dump (
            const char * buf,
            size_t len )  [static]
```

Print given buffer to stdout in hex and bin.

Parameters

buf	Buffer to print
len	Length to print (in buffer)

2.4.3.2 log_event() [1/3]

```
void Logger::log_event (
            clockid_t clock_type,
            const char * format,
            va_list args )
```

Log specified formatted string (printf style) with time of given clock.

Parameters

clock_type	Type of clock which will be used to log
format	Format string (printf style)
args	Arguments for the format string

2.4.3.3 log_event() [2/3]

```
void Logger::log_event (
            clockid_t clock_type,
            const char * format,
             ...  )
```

Log specified formatted string (printf style) with time of given clock.

Parameters

clock_type	Type of clock which will be used to log
format	Format string (printf style)

2.4.3.4 log_event() [3/3]

```
void Logger::log_event (
            const char * format,
             ...  )
```

Log specified formatted string (printf style) with CLOCK_MONOTONIC.

Parameters

format	Format string (printf style)

2.4.3.5 log_time()

```
void Logger::log_time (
            clockid_t clock_type )  [private]
```

Appends time to logging file (without adding endline after)

Parameters

clock_type	Clock to be used

2.4.3.6 print_int_safe()

```
void Logger::print_int_safe (
            int expr )  [static]
```

Prints int to STDOUT (async-signal-safe)

Parameters

expr	Integer to be printed

2.4.3.7 print_string_safe()

```
void Logger::print_string_safe (
            const std::string & expr )  [static]
```

Prints string to STDOUT (async-signal-safe)

Parameters

expr	String to be printed

2.4.3.8 print_time_safe()

```
void Logger::print_time_safe (
            clockid_t clk_id )  [static]
```

Prints time to STDOUT (async-signal-safe)

Parameters

clk_id	Clock to be used

2.4.3.9 push_to_buffer_int_safe()

```
size_t Logger::push_to_buffer_int_safe (
            char * buf,
            int expr )  [static]
```

Appends int to buffer (async-signal-safe)

Parameters

buf	Buffer to which to push to
expr	Integer to be appended

Returns

Number of bytes written

2.4.3.10 push_to_buffer_string_safe()

```
size_t Logger::push_to_buffer_string_safe (
            char * buf,
            const char * expr )  [static]
```

Appends string to buffer (async-signal-safe)

Parameters

buf	Buffer to which to push to
expr	String to be appended

Returns

Number of bytes written

2.4.3.11 push_to_buffer_time_safe()

```
size_t Logger::push_to_buffer_time_safe (
            char * buf,
            clockid_t clk_id )  [static]
```

Appends current time to buffer (async-signal-safe)

Parameters

buf	Buffer to which to push to
clk_id	Clock to be used

Returns

Number of bytes written

The documentation for this class was generated from the following file:

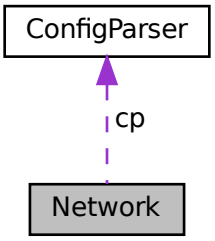
- src/include/logger.hpp

2.5 Network Class Reference

Class responsible for all network control and communication.

```
#include <network.hpp>
```

Collaboration diagram for Network:



Public Member Functions

- Network (const ConfigParser &cp)
  Build TUN interface and set necessary constants.
- struct timespec get_latency (int em_id1, int em_id2) const
  Get latency between process `em_id1` and process `em_id2`.
- struct timespec get_max_latency () const
  Get maximum pairwise latency in the network.
- int get_procs () const
  Get number of processes in the network.
- int get_em_id (const std::string &address) const
  Get emulator's internal id of process with given address.
- std::string get_addr (int em_id) const
  Get address of a process with given internal id.
- std::string get_inter_addr () const
  Get address of TUN interface.
- void send (const Packet &packet) const
  Send the buffer of `packet` object through TUN FD.
- ssize_t receive (char *buffer, size_t buffer_size) const
  Receive data through TUN FD.

Private Member Functions

- void create_tun ()
  Creates and customizes the TUN interface and its subnetwork.

Private Attributes

- int tun_fd
  FD of the TUN interface.
- const ConfigParser & cp
  Configuration read from the file by emulator.
- struct timespec max_latency
  Calculated max pairwise latency.

2.5.1 Detailed Description

Class responsible for all network control and communication.

Class builds and interacts with TUN interface. It sets all necessary interface settings and allows to send and receive packets through TUN file descriptor. This class also allows translation between emulator's internal process id and process address and port in the TUN subnetwork. Additionally the pairwise processes latencies can be read through the functionality provided by this class.

2.5.2 Constructor & Destructor Documentation

2.5.2.1 Network()

```
Network::Network (
            const ConfigParser & cp )
```

Build TUN interface and set necessary constants.

Parameters

cp	Emulator's configuration read from a file

2.5.3 Member Function Documentation

2.5.3.1 get_addr()

```
std::string Network::get_addr (
            int em_id ) const
```

Get address of a process with given internal id.

Parameters

em_id	The id on which to query

Returns

Address (in number/dot form) associated with this internal id

2.5.3.2 get_em_id()

```
int Network::get_em_id (
            const std::string & address ) const
```

Get emulator's internal id of process with given address.

Parameters

address	The address (in number/dot form) on which to query

Returns

Internal id of process associated with this address (-1 if none)

2.5.3.3 get_inter_addr()

```
std::string Network::get_inter_addr ( ) const
```

Get address of TUN interface.

Returns

Address (in number/dot form) of the TUN interface

2.5.3.4 get_latency()

```
struct timespec Network::get_latency (
            int em_id1,
            int em_id2 ) const
```

Get latency between process `em_id1` and process `em_id2`.

Parameters

em_id1	First process
em_id2	Second process

Returns

Pairwise latency between those two processes

2.5.3.5 get_max_latency()

```
struct timespec Network::get_max_latency ( ) const
```

Get maximum pairwise latency in the network.

Returns

Maximum pairwise latency

2.5.3.6 get_procs()

```
int Network::get_procs ( ) const
```

Get number of processes in the network.

Returns

Number of processes in the network

2.5.3.7 receive()

```
ssize_t Network::receive (
            char * buffer,
            size_t buffer_size ) const
```

Receive data through TUN FD.

Parameters

buffer	Placeholder to which received data will be copied
buffer_size	Available place for new data

Returns

Number of received bytes (0 if none)

2.5.3.8 send()

```
void Network::send (
            const Packet & packet ) const
```

Send the buffer of `packet` object through TUN FD.

Parameters

packet	Packet which will be sent

The documentation for this class was generated from the following file:

- src/include/network/network.hpp

2.6 Packet Class Reference

Class encapsulating single ip frame sent through TUN interface.

`#include <packet.hpp>`

Public Member Functions

- Packet (const char *buf, size_t size, struct timespec ts)

  Main packet constructor from data read from TUN FD.
- Packet (const Packet &other)

  Copy constructor.
- Packet (Packet &&other)

  Move constructor.
- Packet & operator= (const Packet &other)

  Assignment operator.
- bool operator< (const Packet &other) const

  Comparison operator (comparison based on timestamp)
- int get_version () const

  Get packet IPv version (4/6)

- size_t **get_size** () const
- char * **get_buffer** () const
- struct timespec **get_ts** () const
- std::string **get_source_addr** () const
- std::string **get_dest_addr** () const
- int **get_source_port** () const
- int **get_dest_port** () const
- void set_source_addr (const std::string &addr)

  Set a new source address for the packet.
- void set_dest_addr (const std::string &addr)

  Set a new destination address for the packet.
- void increase_ts (struct timespec other_ts)

  Increase packet's ts value by `other_ts`.

Private Member Functions

- struct iphdr * **get_iphdr** () const
- struct udphdr * **get_udp** () const
- char * **get_data** () const
- size_t **get_data_len** () const
- uint16_t **ip4_checksum** (const struct iphdr *ip) const
- uint16_t **udp_checksum** (const struct iphdr *ip, const struct udphdr *udp, const char *data, size_t data_len) const
- uint16_t **tcp_checksum** (const struct iphdr *ip, const struct tcphdr *tcp, const char *data, size_t data_len) const
- bool **has_transport_layer_hdr** () const

Private Attributes

- char * buffer

  Buffer in which raw data is stored.
- size_t size

  Size of data stored in the packet.
- struct timespec ts

  Packets timestamp.

2.6.1 Detailed Description

Class encapsulating single ip frame sent through TUN interface.

Instances of this class are moved around the emulator system to keep track of which process messaged which process and at what time. As those instances would be put to different priority_queue all types of constructors need to be implemented. Packets would be sorted on the ts parameter (signifying virtual clock value of the sending process).

It's worth noting, that in the current form packet code works ONLY for IPv4 and UDP protocol.

2.6.2 Constructor & Destructor Documentation

2.6.2.1 Packet() [1/3]

```
Packet::Packet (
            const char * buf,
            size_t size,
            struct timespec ts )
```

Main packet constructor from data read from TUN FD.

Constructors copies data from the `buf` to newly allocated buffer which is freed in destructor

Parameters

buf	Char buffer read from TUN FD
size	Number of bytes read
ts	Virtual clock value of the sending process

2.6.2.2 Packet() [2/3]

```
Packet::Packet (
            const Packet & other )
```

Copy constructor.

Parameters

other	Packet to copy from

2.6.2.3 Packet() [3/3]

```
Packet::Packet (
            Packet && other )
```

Move constructor.

Parameters

other	Packet which insides will be moved here

2.6.3 Member Function Documentation

2.6.3.1 get_version()

```
int Packet::get_version ( ) const
```

Get packet IPv version (4/6)

Its worth noting that some of packets functionalities don't work for IPv6. Especially setting source or destination addresses are specifically implemented for IPv4

Returns

Packet IPv version (4/6)

2.6.3.2 increase_ts()

```
void Packet::increase_ts (
            struct timespec other_ts )
```

Increase packet's ts value by `other_ts`.

Parameters

other_ts	Time by which to increate packet's ts value

2.6.3.3 operator<()

```
bool Packet::operator< (
            const Packet & other ) const
```

Comparison operator (comparison based on timestamp)

Parameters

other	Packet to compare this on with

2.6.3.4 operator=()

```
Packet & Packet::operator= (
            const Packet & other )
```

Assignment operator.

Parameters

other	Original packet

2.6.3.5 set_dest_addr()

```
void Packet::set_dest_addr (
            const std::string & addr )
```

Set a new destination address for the packet.

Changes packets buffer value so that the destination address is updated. This requires the IPv4 checksum to be updated, as well as the UDP checksum in udphdr to be updated.

Parameters

addr	New destination address (in number/dot form)

2.6.3.6 set_source_addr()

```
void Packet::set_source_addr (
            const std::string & addr )
```

Set a new source address for the packet.

Changes packets buffer value so that the source address is updated. This requires the IPv4 checksum to be updated, as well as the UDP checksum in udphdr to be updated.

Parameters

addr	New source address (in number/dot form)

The documentation for this class was generated from the following file:

- src/include/network/packet.hpp

Index

SimpleEM Documentation

Tinyem Part

Generated by Doxygen 1.9.1

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

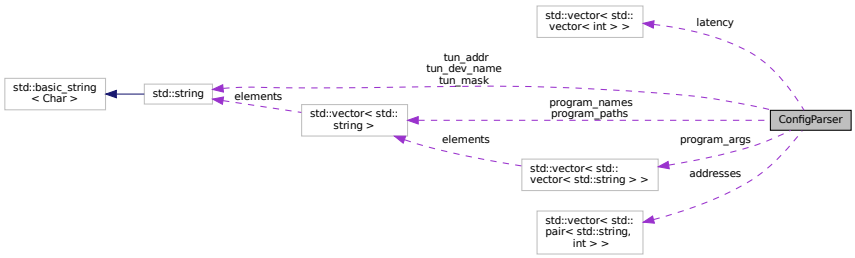
Class Documentation

3.1 ConfigParser Class Reference

Class parsing and saving the configuration from a file.

`#include <config-parser.hpp>`

Collaboration diagram for ConfigParser:



Public Member Functions

- ConfigParser (const std::string &config_path)

  Reads config from the config file (`config_path`)

Public Attributes

- std::string tun_dev_name

  Device name for new TUN interface.
- std::string tun_addr

  Address of the TUN interface.
- std::string tun_mask

  Mask of the TUN interface subnetwork.
- int procs

  Number of processes to be emulated by the emulator.

- std::vector< std::vector< int > > latency

  Matrix of pairwise latencies.
- std::vector< std::pair< std::string, int > > addresses

  Addresses (in number/dot format) and ports of processes.
- std::vector< std::string > program_paths

  Paths to programs to be run on every process.
- std::vector< std::string > program_names

  Names of programs to be run on every process.
- std::vector< std::vector< std::string > > program_args

  Arguments to be passed to every process.

3.1.1 Detailed Description

Class parsing and saving the configuration from a file.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 ConfigParser()

```
ConfigParser::ConfigParser (
            const std::string & config_path )
```

Reads config from the config file (`config_path`)

Config layout is as follows:

- First line consists of three words split by whitespace, tun device name, tun interface address, and tun interface address mask, to be used in setting up the tun interface.
- Second line consists of one number - procs - num of procs to simulate
- Next procs lines consists of a string and int each, i-th line means the address and port on which i-th proc is listening
- Next procs lines consists of procs numbers each, i-th number in j-th line means the latency from i-th to j-th proc in miliseconds (i-th number in i-th column should be 0)
- Next procs lines consist of at least two words each, i-th line starts with the path to i-th program, then the name of the i-th program and then whitespace-split args to the i-th program.

3.1.3 Member Data Documentation

3.1.3.1 addresses

```
std::vector<std::pair<std::string, int> > ConfigParser::addresses
```

Addresses (in number/dot format) and ports of processes.

3.1.3.2 latency

```
std::vector<std::vector<int> > ConfigParser::latency
```

Matrix of pairwise latencies.

3.1.3.3 procs

```
int ConfigParser::procs
```

Number of processes to be emulated by the emulator.

3.1.3.4 program_args

```
std::vector<std::vector<std::string> > ConfigParser::program_args
```

Arguments to be passed to every process.

3.1.3.5 program_names

```
std::vector<std::string> ConfigParser::program_names
```

Names of programs to be run on every process.

3.1.3.6 program_paths

```
std::vector<std::string> ConfigParser::program_paths
```

Paths to programs to be run on every process.

3.1.3.7 tun_addr

```
std::string ConfigParser::tun_addr
```

Address of the TUN interface.

3.1.3.8 tun_dev_name

```
std::string ConfigParser::tun_dev_name
```

Device name for new TUN interface.

3.1.3.9 tun_mask

```
std::string ConfigParser::tun_mask
```

Mask of the TUN interface subnetwork.

The documentation for this class was generated from the following file:

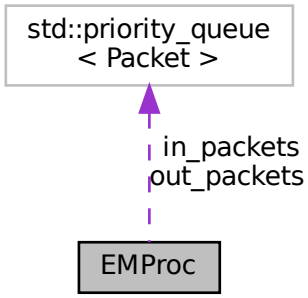
- config-parser.hpp

3.2 EMProc Class Reference

Controller of a single process inside an emulation.

```
#include <emproc.hpp>
```

Collaboration diagram for EMProc:



Public Member Functions

- EMProc (em_id_t em_id, int pid)

  Class main constructor.
- void awake (struct timespec ts, const Network &network)

  Awake emulated process and let him run for specified amount of time, intercepting packets sent by it.

Public Attributes

- em_id_t em_id

  Internal id of emulated process.
- int pid

  pid of emulated process
- struct timespec virtual_clock

  Time that the process was awake.
- std::priority_queue< Packet > out_packets

  Buffer of packets sent by process.
- std::priority_queue< Packet > in_packets

  Buffer of packets to be received by process.

Private Member Functions

- bool to_receive_before (struct timespec ts)

  Check if there is any packet that this process should receive before the specified timestamp.

3.2.1 Detailed Description

Controller of a single process inside an emulation.

Class responsible for the state and awakening of a single process in an emulation. It holds the queues of packets that were sent from the process as well as the queue of packets that should be delivered to the process. It keeps track of process virtual clock. All the inside-awakening logic is kept in this class.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 EMProc()

```
EMProc::EMProc (
            em_id_t em_id,
            int pid )  [inline]
```

Class main constructor.

Sets process' virtual clock to 0.

Parameters

em↩ _id	Emulator's internal process id of this process
pid	Operating systems pid of process associated with this emulated process

3.2.3 Member Function Documentation

3.2.3.1 awake()

```
void EMProc::awake (
            struct timespec ts,
            const Network & network )
```

Awake emulated process and let him run for specified amount of time, intercepting packets sent by it.

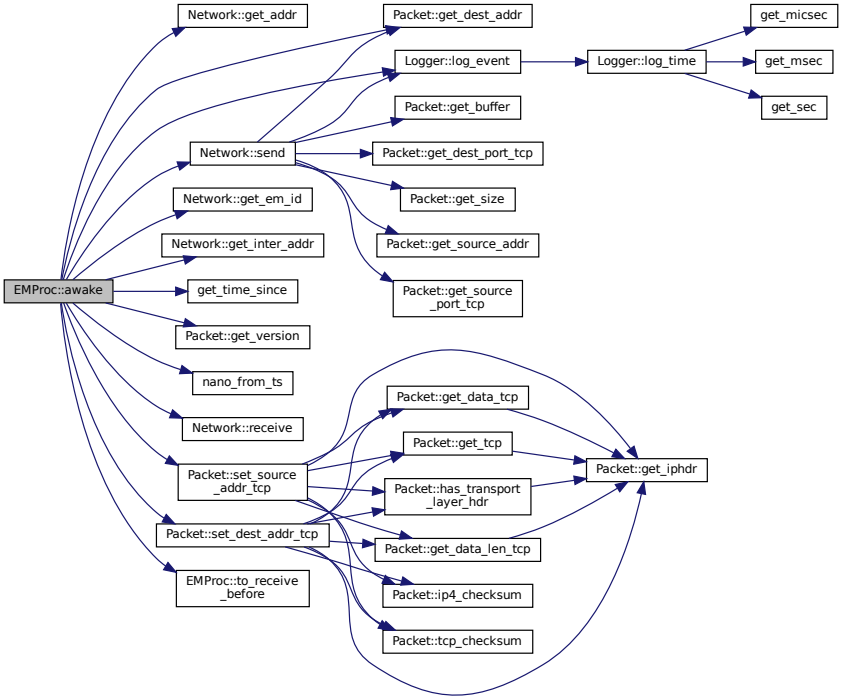
Awakes emulated process for amount of time specified by `ts` . All packets sent by this process to addresses in the subnetwork of TUN interface specified by `tun_fd` will be intercepted and stored in out_packets.

Function sends `SIGCONT` signal to specified process, then after `ts` time it sends the `SIGSTOP` signal. During the time that the process is running all packets sent by it will be intercepted. Also, in appropriate times, packets from in_packets will be sent to the process using the TUN interface with `tun_fd` . virtual_clock is updated inside this function. It is guaranteed that when the function returns, the specified process has already stopped.

Parameters

ts	Time for the process to run
network	Network on which the simulator is operating

Here is the call graph for this function:



3.2.3.2 to_receive_before()

```
bool EMProc::to_receive_before (
            struct timespec ts )  [private]
```

Check if there is any packet that this process should receive before the specified timestamp.

Checks if the first packet in in_packets buffer exists and has timestamp lower then the one specified by `ts`

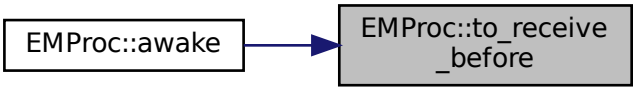
Parameters

ts	Time for which to check

Returns

If such packet exists

Here is the caller graph for this function:



3.2.4 Member Data Documentation

3.2.4.1 em_id

```
em_id_t EMProc::em_id
```

Internal id of emulated process.

3.2.4.2 in_packets

```
std::priority_queue<Packet> EMProc::in_packets
```

Buffer of packets to be received by process.

3.2.4.3 out_packets

```
std::priority_queue<Packet> EMProc::out_packets
```

Buffer of packets sent by process.

3.2.4.4 pid

```
int EMProc::pid
```

pid of emulated process

3.2.4.5 virtual_clock

`struct timespec EMProc::virtual_clock`

Time that the process was awake.

The documentation for this class was generated from the following file:

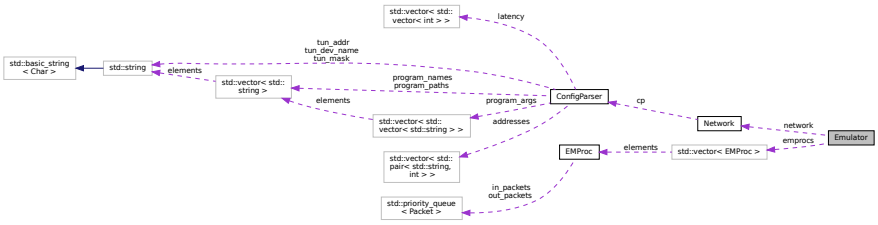
- emproc.hpp

3.3 Emulator Class Reference

Class encapsulating the main logic of the emulator.

`#include <emulator.hpp>`

Collaboration diagram for Emulator:



Public Member Functions

- Emulator (Network &network, const ConfigParser &cp)

  Creates an emulation and spawns specified number of new processes.
- void start_emulation (int steps)

  Starts the emulation by iteratively scheduling processes.
- void kill_emulation ()

  Kills all spawned processes.

Private Member Functions

- void fork_stop_run (int *pids, const ConfigParser &cp)

  Forks the process and initializes child processes.
- const char * extractProgramName (const char *filePath)

  Extracts the name of the program from its path.
- void executeProgram (const std::string &program_path, const std::vector< std::string > &program_args)

  Executes the program specified by `program_path`.
- void child_init (const std::string &program_path, const std::vector< std::string > &program_args)

  Initialize the child process.
- em_id_t choose_next_proc () const

  Chooses next process to be scheduled for awakening.
- struct timespec get_time_interval (em_id_t em_id) const

  Returns the longest time `em_id` can run without violating correctness.
- void schedule_sent_packets (em_id_t em_id)

  Moves packets sent by `em_id` to appropriate in queues of receiving processes.

Private Attributes

- int procs

  Number of processes being emulated.
- std::vector< EMProc > emprocs

  States of each process.
- Network & network

  Specifies network on which the emulation is being run.

3.3.1 Detailed Description

Class encapsulating the main logic of the emulator.

Class responsible for overall control of the emulation. It creates specified number of new processes and runs the emulated program on them. It controls the state of those processes (whether they are stopped or running), and decides which one should be awaken (scheduled) next and for how lond. At last, it is responsible for the cleanup after the emulation is finished.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 Emulator()

```
Emulator::Emulator (
            Network & network,
            const ConfigParser & cp )
```

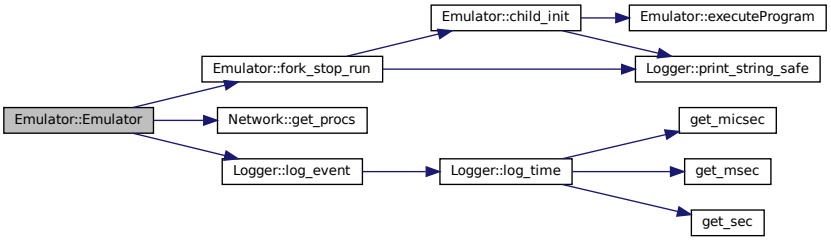
Creates an emulation and spawns specified number of new processes.

Creates the emulation by forking the process procs times. Every newly created process immediately stops itself by the `raise(SIGSTOP)` call. Objects corresponding to newly created processes are stored in the emprocs vector.

Parameters

network	The network on which the emulator runs
cp	Configuration of the emulation

Here is the call graph for this function:



3.3.3 Member Function Documentation

3.3.3.1 child_init()

```
void Emulator::child_init (
            const std::string & program_path,
            const std::vector< std::string > & program_args )  [private]
```

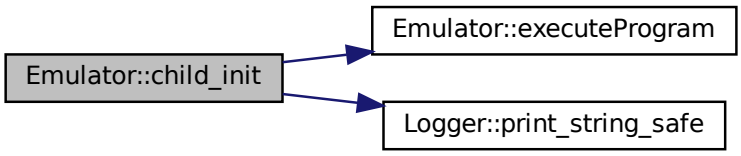
Initialize the child process.

Start by doing the `raise(SIGSTOP)` call. After the process is awaken (using the `SIGCONT` signal), it executes the program specified by `program_path`.

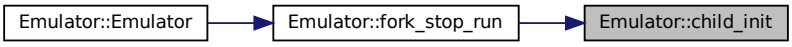
Parameters

program_path	Path to the program to be executed on this process
program_name	Name of the program to be executed on this process
program_args	Arguments to be passes to the program

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.3.2 choose_next_proc()

```
em_id_t Emulator::choose_next_proc ( ) const  [private]
```

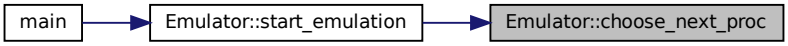
Chooses next process to be scheduled for awakening.

Loops through all te processes and choses the one which was executed for the least amount of time.

Returns

Id of the process to be scheduled next

Here is the caller graph for this function:



3.3.3.3 executeProgram()

```
void Emulator::executeProgram (
            const std::string & program_path,
            const std::vector< std::string > & program_args )  [private]
```

Executes the program specified by `program_path`.

Executes the program specified by `program_path` with arguments specified by `program_args`. The program is executed using the `system()` call.

Parameters

program_path	Path to the program to be executed
program_args	Arguments to be passed to the program

Here is the caller graph for this function:

3.3.3.4 extractProgramName()

```
const char * Emulator::extractProgramName (
            const char * filePath )  [private]
```

Extracts the name of the program from its path.

Extracts the name of the program from its path. If the path contains '/' character, the name is everything after the last occurence of that character. Otherwise, the name is the entire path.

Parameters

filePath	Path to the program

Returns

Name of the program

3.3.3.5 fork_stop_run()

```
void Emulator::fork_stop_run (
            int * pids,
            const ConfigParser & cp )  [private]
```

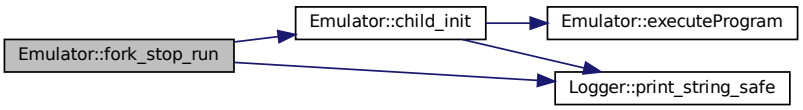
Forks the process and initializes child processes.

Creates procs new processes that execute the child_init function. Saves process ids of newly created processes in the `pids` array.

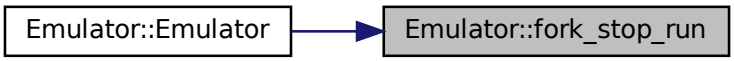
Parameters

pids	Array in which to store process ids of new processes
cp	Configuration to be used for new processes initialization

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.3.6 get_time_interval()

```
struct timespec Emulator::get_time_interval (
            em_id_t em_id ) const  [private]
```

Returns the longest time `em_id` can run without violating correctness.

Every process has its virtual clock saved, returns minimum (over all processes p != `em_id`) of
`p->virtual_clock - em_id->virtual_clock + network.get_latency(p, em_id)`

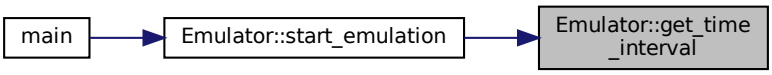
Parameters

em↩ _id	Process which will be run

Returns

The maximum possible time to run

Here is the caller graph for this function:



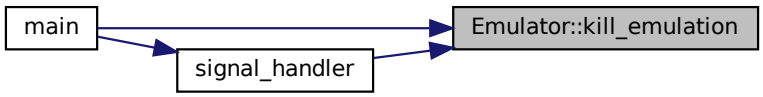
3.3.3.7 kill_emulation()

```
void Emulator::kill_emulation ( )
```

Kills all spawned processes.

Kills all spawned processes, effectively ending the emulation. Here is the call graph for this function:

Here is the caller graph for this function:



3.3.3.8 schedule_sent_packets()

```
void Emulator::schedule_sent_packets (
            em_id_t em_id ) [private]
```

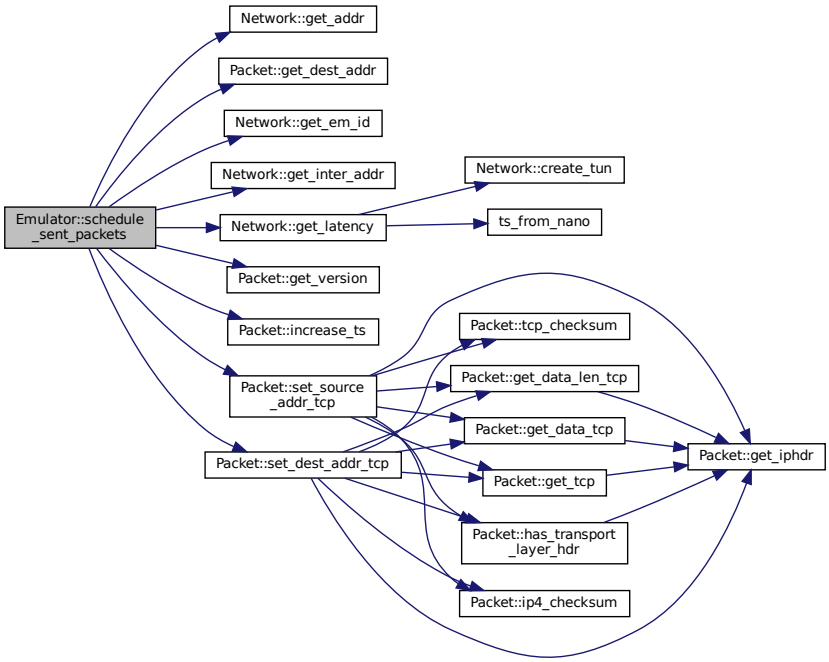
Moves packets sent by `em_id` to appropriate in queues of receiving processes.

For every packet sent by the `em_id`, this function moves it to a queue of packets awaiting to be received by appropriate other process. The function increases the timestamp of the packet by the pairwise latency between those two processes - so that the packet will be received at proper time.

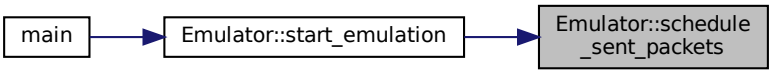
Parameters

em↩ _id	Id of the process whose out packets need to be moved

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.3.9 start_emulation()

```
void Emulator::start_emulation (
            int steps )
```

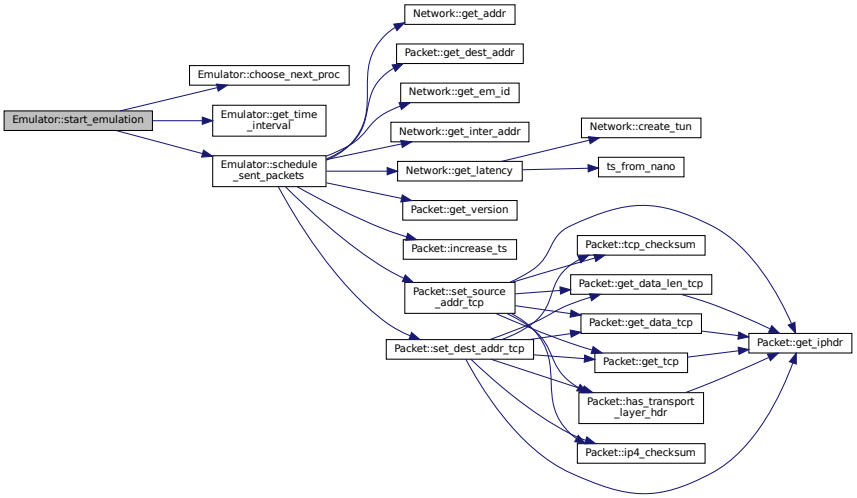
Starts the emulation by iteratively scheduling processes.

Performs `steps` times the loop of choosing the next process to schedule, calculating for what time it should run, awakening it for that time, and later sorting packets sent by it to appropriate queues.

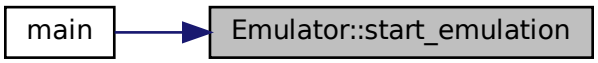
Parameters

steps	Number of awakenings to perform

Here is the call graph for this function:



Here is the caller graph for this function:



3.3.4 Member Data Documentation

3.3.4.1 emprocs

```
std::vector<EMProc> Emulator::emprocs  [private]
```

States of each process.

3.3.4.2 network

```
Network& Emulator::network  [private]
```

Specifies network on which the emulation is being run.

3.3.4.3 procs

```
int Emulator::procs  [private]
```

Number of processes being emulated.

The documentation for this class was generated from the following file:

- emulator.hpp

3.4 Logger Class Reference

Utility class for logging in the system.

```
#include <logger.hpp>
```

Public Member Functions

- Logger (const std::string &file_path)
  Initializes logger and opens specified file.
- ~Logger ()
- void log_event (clockid_t clock_type, const char *format, va_list args)
  Log specified formatted string (printf style) with time of given clock.
- void log_event (clockid_t clock_type, const char *format,...)
  Log specified formatted string (printf style) with time of given clock.
- void log_event (const char *format,...)
  Log specified formatted string (printf style) with CLOCK_MONOTONIC.

Static Public Member Functions

- static void dump (const char *buf, size_t len)
  Print given buffer to stdout in hex and bin.
- static size_t push_to_buffer_time_safe (char *buf, clockid_t clk_id)
  Appends current time to buffer (async-signal-safe)
- static size_t push_to_buffer_string_safe (char *buf, const char *expr)
  Appends string to buffer (async-signal-safe)
- static size_t push_to_buffer_int_safe (char *buf, int expr)
  Appends int to buffer (async-signal-safe)
- static void print_string_safe (const std::string &expr)
  Prints string to STDOUT (async-signal-safe)
- static void print_int_safe (int expr)
  Prints int to STDOUT (async-signal-safe)
- static void print_time_safe (clockid_t clk_id)
  Prints time to STDOUT (async-signal-safe)

Private Member Functions

- void log_time (clockid_t clock_type)
  Appends time to logging file (without adding endline after)

Private Attributes

- FILE * file_ptr

  Pointer to FILE object to which to write.

3.4.1 Detailed Description

Utility class for logging in the system.

Logger is used by the emulator for logging all recorded events in a structured manner.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Logger()

```
Logger::Logger (
            const std::string & file_path )
```

Initializes logger and opens specified file.

Parameters

file_path	Path to file to which to log to

3.4.2.2 ~Logger()

```
Logger::~Logger ( )
```

3.4.3 Member Function Documentation

3.4.3.1 dump()

```
void Logger::dump (
            const char * buf,
            size_t len )  [static]
```

Print given buffer to stdout in hex and bin.

Parameters

buf	Buffer to print
len	Length to print (in buffer)

3.4.3.2 log_event() [1/3]

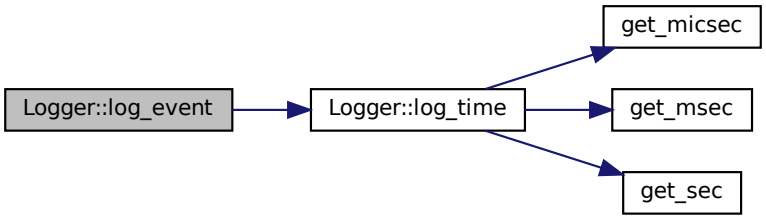
```
void Logger::log_event (
            clockid_t clock_type,
            const char * format,
            va_list args )
```

Log specified formatted string (printf style) with time of given clock.

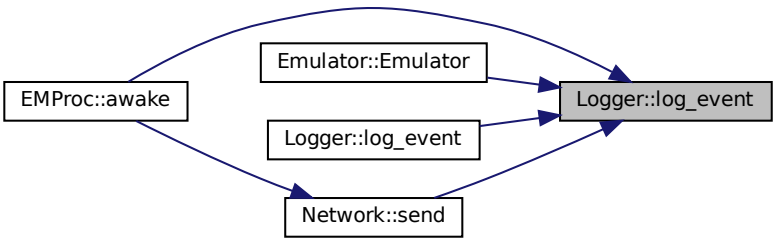
Parameters

clock_type	Type of clock which will be used to log
format	Format string (printf style)
args	Arguments for the format string

Here is the call graph for this function:



Here is the caller graph for this function:



3.4.3.3 log_event() [2/3]

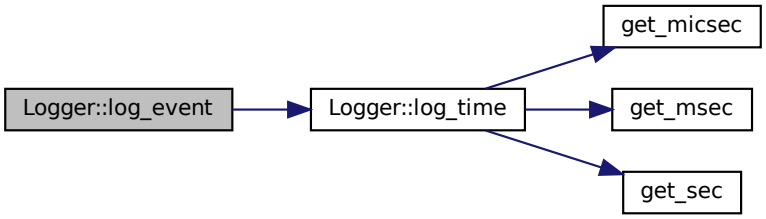
```
void Logger::log_event (
            clockid_t clock_type,
            const char * format,
             ...  )
```

Log specified formatted string (printf style) with time of given clock.

Parameters

clock_type	Type of clock which will be used to log
format	Format string (printf style)

Here is the call graph for this function:



3.4.3.4 log_event() [3/3]

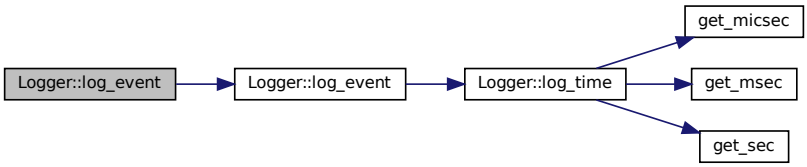
```
void Logger::log_event (
            const char * format,
             ...  )
```

Log specified formatted string (printf style) with CLOCK_MONOTONIC.

Parameters

format	Format string (printf style)

Here is the call graph for this function:



3.4.3.5 log_time()

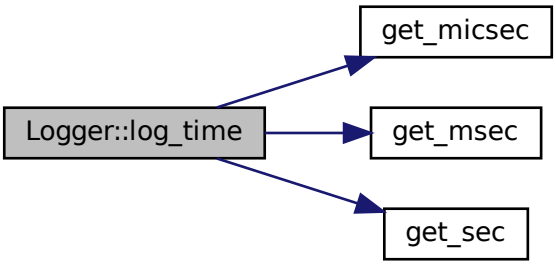
```
void Logger::log_time (
            clockid_t clock_type )  [private]
```

Appends time to logging file (without adding endline after)

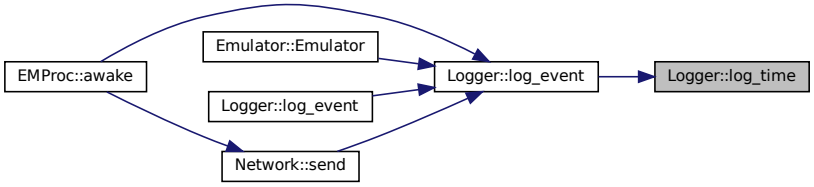
Parameters

clock_type	Clock to be used

Here is the call graph for this function:



Here is the caller graph for this function:



3.4.3.6 print_int_safe()

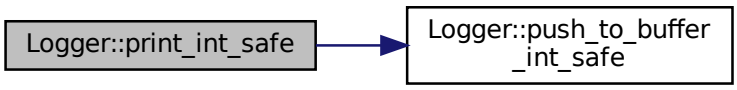
```
void Logger::print_int_safe (
            int expr )  [static]
```

Prints int to STDOUT (async-signal-safe)

Parameters

expr	Integer to be printed

Here is the call graph for this function:



3.4.3.7 print_string_safe()

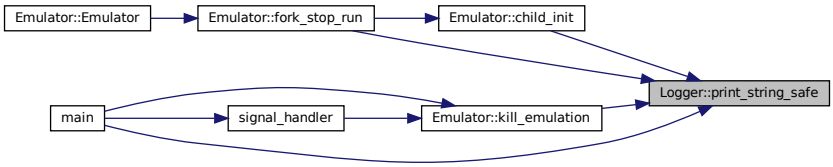
```
void Logger::print_string_safe (
            const std::string & expr )  [static]
```

Prints string to STDOUT (async-signal-safe)

Parameters

expr	String to be printed

Here is the caller graph for this function:



3.4.3.8 print_time_safe()

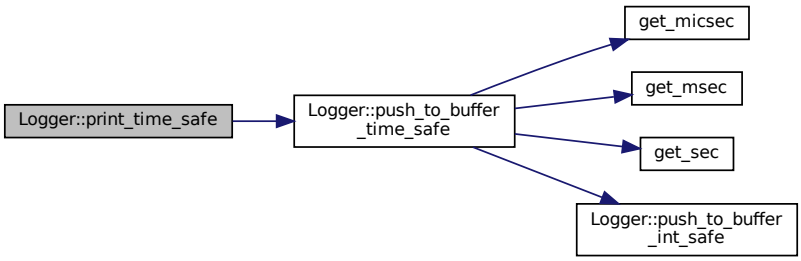
```
void Logger::print_time_safe (
            clockid_t clk_id )  [static]
```

Prints time to STDOUT (async-signal-safe)

Parameters

clk_id	Clock to be used

Here is the call graph for this function:



3.4.3.9 push_to_buffer_int_safe()

```
size_t Logger::push_to_buffer_int_safe (
            char * buf,
            int expr )  [static]
```

Appends int to buffer (async-signal-safe)

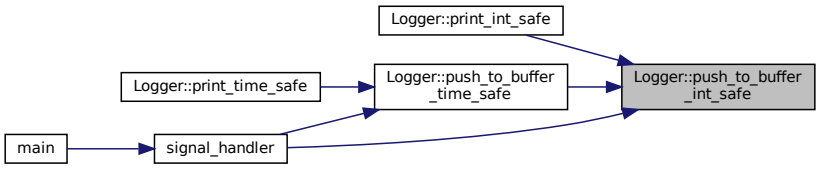
Parameters

buf	Buffer to which to push to
expr	Integer to be appended

Returns

Number of bytes written

Here is the caller graph for this function:



3.4.3.10 push_to_buffer_string_safe()

```
size_t Logger::push_to_buffer_string_safe (
            char * buf,
            const char * expr )  [static]
```

Appends string to buffer (async-signal-safe)

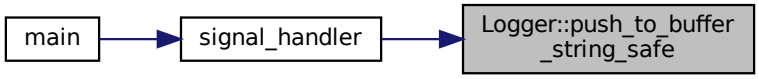
Parameters

buf	Buffer to which to push to
expr	String to be appended

Returns

Number of bytes written

Here is the caller graph for this function:



3.4.3.11 push_to_buffer_time_safe()

```
size_t Logger::push_to_buffer_time_safe (
            char * buf,
            clockid_t clk_id )  [static]
```

Appends current time to buffer (async-signal-safe)

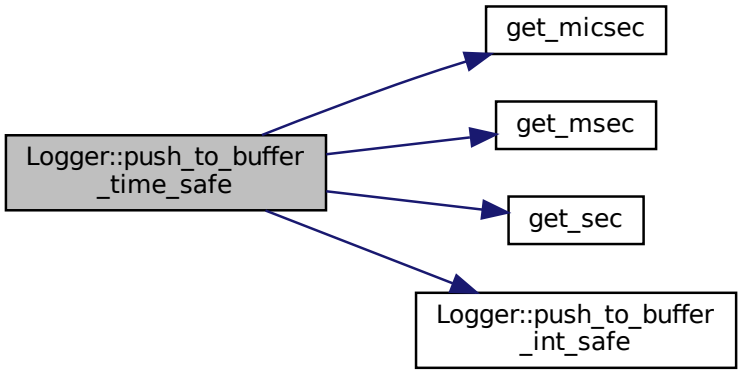
Parameters

buf	Buffer to which to push to
clk_id	Clock to be used

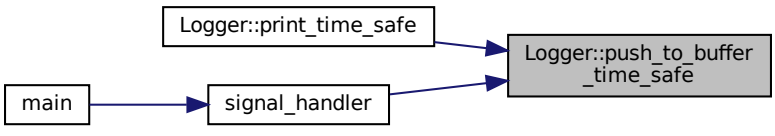
Returns

Number of bytes written

Here is the call graph for this function:



Here is the caller graph for this function:



3.4.4 Member Data Documentation

3.4.4.1 file_ptr

```
FILE* Logger::file_ptr  [private]
```

Pointer to FILE object to which to write.

The documentation for this class was generated from the following file:

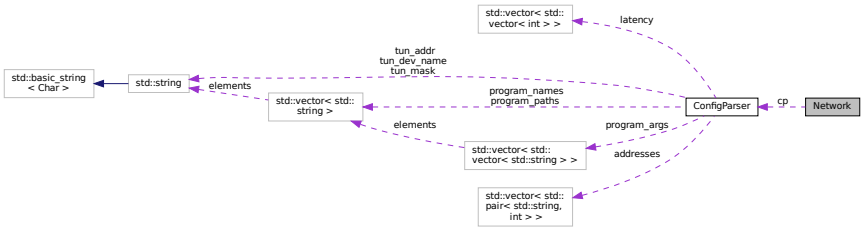
- logger.hpp

3.5 Network Class Reference

Class responsible for all network control and communication.

`#include <network.hpp>`

Collaboration diagram for Network:



Public Member Functions

- Network (const ConfigParser &cp)

  Build TUN interface and set necessary constants.
- struct timespec get_latency (int em_id1, int em_id2) const

  Get latency between process `em_id1` and process `em_id2`.
- struct timespec get_max_latency () const

  Get maximum pairwise latency in the network.
- int get_procs () const

  Get number of processes in the network.
- int get_em_id (const std::string &address) const

  Get emulator's internal id of process with given address.
- std::string get_addr (int em_id) const

  Get address of a process with given internal id.
- std::string get_inter_addr () const

  Get address of TUN interface.
- void send (const Packet &packet) const

  Send the buffer of `packet` object through TUN FD.
- ssize_t receive (char *buffer, size_t buffer_size) const

  Receive data through TUN FD.

Private Member Functions

- void create_tun ()

  Creates and customizes the TUN interface and its subnetwork.

Private Attributes

- int tun_fd

  FD of the TUN interface.
- const ConfigParser & cp

  Configuration read from the file by emulator.
- struct timespec max_latency

  Calculated max pairwise latency.

3.5.1 Detailed Description

Class responsible for all network control and communication.

Class builds and interacts with TUN interface. It sets all necessary interface settings and allows to send and receive packets through TUN file descriptor. This class also allows translation between emulator's internal process id and process address and port in the TUN subnetwork. Additionally the pairwise processes latencies can be read through the functionality provided by this class.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Network()

```
Network::Network (
            const ConfigParser & cp )
```

Build TUN interface and set necessary constants.

Parameters

cp	Emulator's configuration read from a file

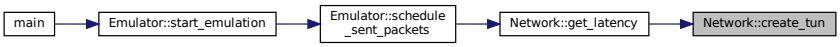
3.5.3 Member Function Documentation

3.5.3.1 create_tun()

```
void Network::create_tun ( )  [private]
```

Creates and customizes the TUN interface and its subnetwork.

Here is the caller graph for this function:



3.5.3.2 get_addr()

```
std::string Network::get_addr (
            int em_id ) const
```

Get address of a process with given internal id.

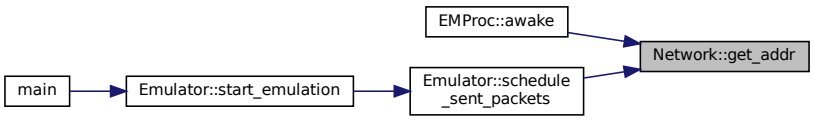
Parameters

em↩ _id	The id on which to query

Returns

Address (in number/dot form) associated with this internal id

Here is the caller graph for this function:



3.5.3.3 get_em_id()

```
int Network::get_em_id (
            const std::string & address ) const
```

Get emulator's internal id of process with given address.

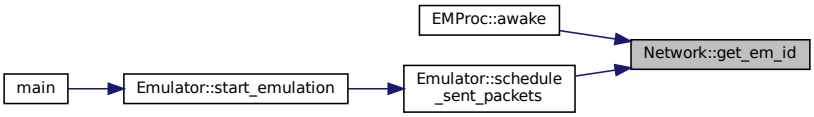
Parameters

address	The address (in number/dot form) on which to query

Returns

Internal id of process associated with this address (-1 if none)

Here is the caller graph for this function:



3.5.3.4 get_inter_addr()

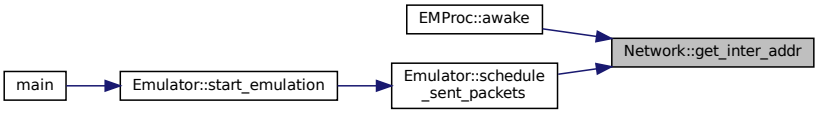
```
std::string Network::get_inter_addr ( ) const
```

Get address of TUN interface.

Returns

Address (in number/dot form) of the TUN interface

Here is the caller graph for this function:



3.5.3.5 get_latency()

```
struct timespec Network::get_latency (
            int em_id1,
            int em_id2 ) const
```

Get latency between process `em_id1` and process `em_id2`.

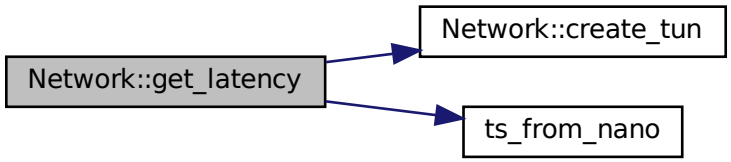
Parameters

em_id1	First process
em_id2	Second process

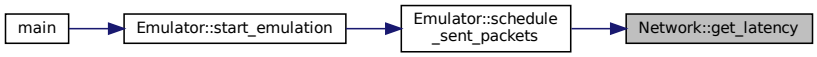
Returns

Pairwise latency between those two processes

Here is the call graph for this function:



Here is the caller graph for this function:



3.5.3.6 get_max_latency()

```
struct timespec Network::get_max_latency ( ) const
```

Get maximum pairwise latency in the network.

Returns

Maximum pairwise latency

3.5.3.7 get_procs()

```
int Network::get_procs ( ) const
```

Get number of processes in the network.

Returns

Number of processes in the network

Here is the caller graph for this function:

3.5.3.8 receive()

```
ssize_t Network::receive (
            char * buffer,
            size_t buffer_size ) const
```

Receive data through TUN FD.

Parameters

buffer	Placeholder to which received data will be copied
buffer_size	Available place for new data

Returns

Number of received bytes (0 if none)

Here is the caller graph for this function:

3.5.3.9 send()

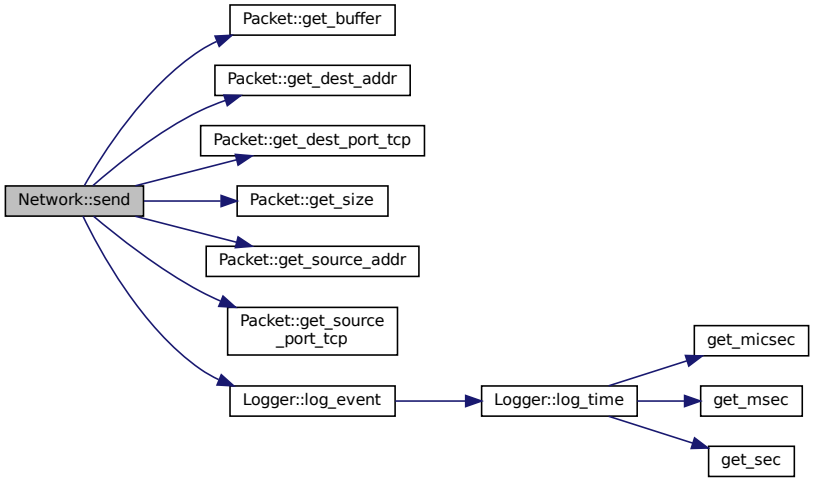
```
void Network::send (
            const Packet & packet ) const
```

Send the buffer of `packet` object through TUN FD.

Parameters

packet	Packet which will be sent

Here is the call graph for this function:



Here is the caller graph for this function:

3.5.4 Member Data Documentation

3.5.4.1 cp

```
const ConfigParser& Network::cp  [private]
```

Configuration read from the file by emulator.

3.5.4.2 max_latency

```
struct timespec Network::max_latency  [private]
```

Calculated max pairwise latency.

3.5.4.3 tun_fd

```
int Network::tun_fd  [private]
```

FD of the TUN interface.

The documentation for this class was generated from the following file:

- network.hpp

3.6 Packet Class Reference

Class encapsulating single ip frame sent through TUN interface.

```
#include <packet.hpp>
```

Public Member Functions

- Packet (const char *buf, size_t size, struct timespec ts)
  Main packet constructor from data read from TUN FD.
- Packet (const Packet &other)
  Copy constructor.
- Packet (Packet &&other)
  Move constructor.
- Packet & operator= (const Packet &other)
  Assignment operator.
- bool operator< (const Packet &other) const
  Comparison operator (comparison based on timestamp)
- ~Packet ()
- int get_version () const
  Get packet IPv version (4/6)
- size_t get_size () const

Get packet size (private variable)

- char * get_buffer () const

  Get packet buffer (private variable)

- struct timespec get_ts () const

  Get packet timestamp (private variable)

- std::string get_source_addr () const

  Get packet source address (from IPv4 header)

- std::string get_dest_addr () const

  Get packet destination address (from IPv4 header)

- int get_source_port () const

  Get packet source port (from UDP header)

- int get_dest_port () const

  Get packet destination port (from UDP header)

- int get_source_port_tcp () const

  Get packet source port (from TCP header)

- int get_dest_port_tcp () const

  Get packet destination port (from TCP header)

- int get_tcp_checksum () const

  Get packet TCP checksum (from TCP header)

- void dump ()

  Dump packet contents in hex format to stdout.

- void set_source_addr (const std::string &addr)

  Set a new source address for the packet.

- void set_dest_addr (const std::string &addr)

  Set a new destination address for the packet.

- void set_source_addr_tcp (const std::string &addr)

  Set a new source address for the packet in TCP.

- void set_dest_addr_tcp (const std::string &addr)

  Set a new destination address for the packet in TCP.

- void increase_ts (struct timespec other_ts)

  Increase packet's ts value by `other_ts`*.*

Private Member Functions

- struct iphdr * get_iphdr () const
- struct udphdr * get_udp () const
- struct tcphdr * get_tcp () const
- char * get_data () const
- size_t get_data_len () const
- char * get_data_tcp () const
- size_t get_data_len_tcp () const
- uint16_t ip4_checksum (const struct iphdr *ip) const
- uint16_t udp_checksum (const struct iphdr *ip, const struct udphdr *udp, const char *data, size_t data_len) const
- uint16_t tcp_checksum (const struct iphdr *ip, const struct tcphdr *tcp, const char *data, size_t data_len) const
- bool has_transport_layer_hdr () const

Private Attributes

- char * buffer

  Buffer in which raw data is stored.
- size_t size

  Size of data stored in the packet.
- struct timespec ts

  Packets timestamp.

3.6.1 Detailed Description

Class encapsulating single ip frame sent through TUN interface.

Instances of this class are moved around the emulator system to keep track of which process messaged which process and at what time. As those instances would be put to different priority_queue all types of constructors need to be implemented. Packets would be sorted on the ts parameter (signifying virtual clock value of the sending process).

It's worth noting, that in the current form packet code works ONLY for IPv4 and UDP protocol.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Packet() [1/3]

```
Packet::Packet (
            const char * buf,
            size_t size,
            struct timespec ts )
```

Main packet constructor from data read from TUN FD.

Constructors copies data from the `buf` to newly allocated buffer which is freed in destructor

Parameters

buf	Char buffer read from TUN FD
size	Number of bytes read
ts	Virtual clock value of the sending process

3.6.2.2 Packet() [2/3]

```
Packet::Packet (
            const Packet & other )
```

Copy constructor.

Parameters

other	Packet to copy from

3.6.2.3 Packet() [3/3]

```
Packet::Packet (
            Packet && other )
```

Move constructor.

Parameters

other	Packet which insides will be moved here

3.6.2.4 ~Packet()

```
Packet::~Packet ( )
```

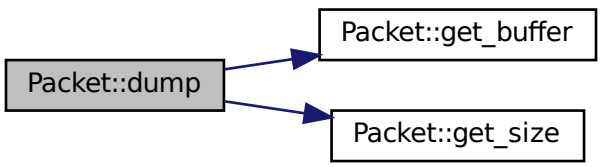
3.6.3 Member Function Documentation

3.6.3.1 dump()

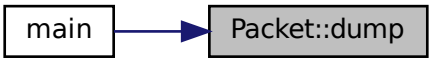
```
void Packet::dump ( )
```

Dump packet contents in hex format to stdout.

Here is the call graph for this function:



Here is the caller graph for this function:



3.6.3.2 get_buffer()

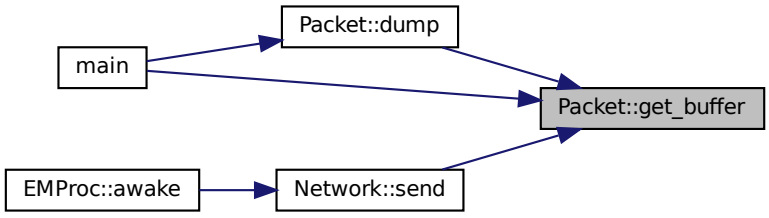
```
char * Packet::get_buffer ( ) const
```

Get packet buffer (private variable)

Returns

Packet buffer

Here is the caller graph for this function:

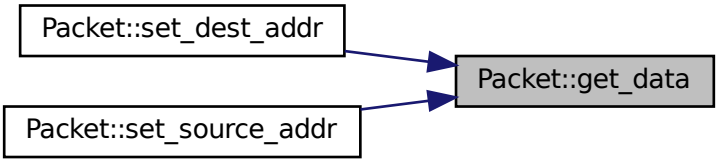


3.6.3.3 get_data()

```
char * Packet::get_data ( ) const  [private]
```

Here is the call graph for this function:

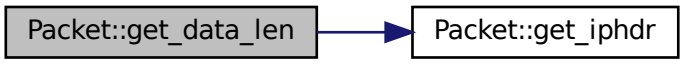
Here is the caller graph for this function:



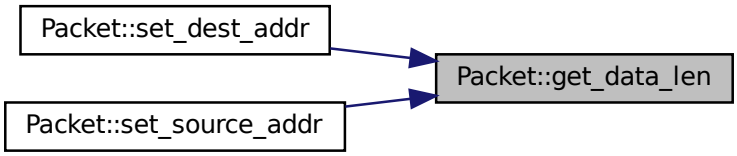
3.6.3.4 get_data_len()

```
size_t Packet::get_data_len ( ) const  [private]
```

Here is the call graph for this function:



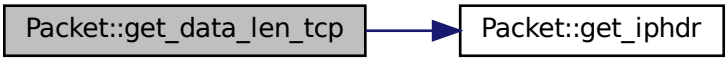
Here is the caller graph for this function:



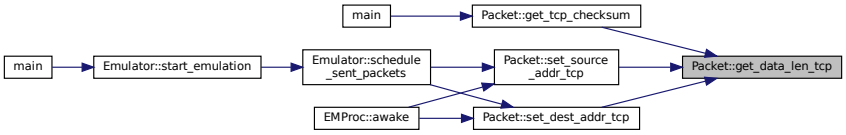
3.6.3.5 get_data_len_tcp()

```
size_t Packet::get_data_len_tcp ( ) const  [private]
```

Here is the call graph for this function:



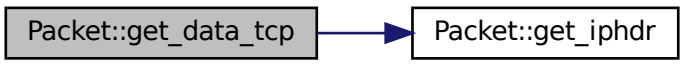
Here is the caller graph for this function:



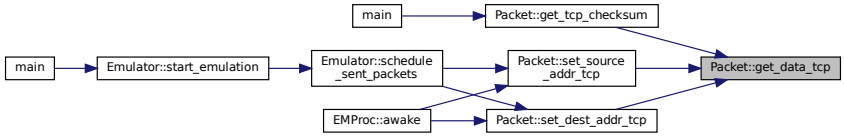
3.6.3.6 get_data_tcp()

```
char * Packet::get_data_tcp ( ) const  [private]
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.6.3.7 get_dest_addr()

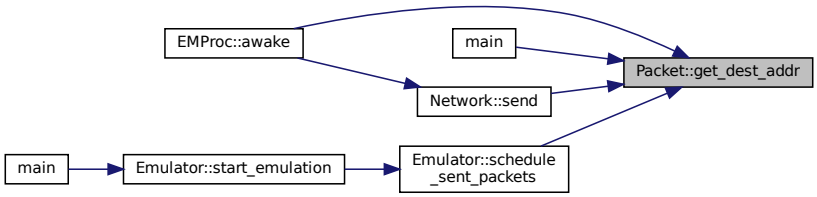
```
std::string Packet::get_dest_addr ( ) const
```

Get packet destination address (from IPv4 header)

Returns

Packet destination address (in number/dot form)

Here is the caller graph for this function:



3.6.3.8 get_dest_port()

```
int Packet::get_dest_port ( ) const
```

Get packet destination port (from UDP header)

Returns

Packet destination port

Here is the caller graph for this function:

3.6.3.9 get_dest_port_tcp()

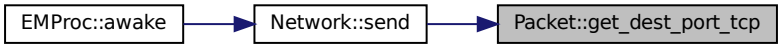
```
int Packet::get_dest_port_tcp ( ) const
```

Get packet destination port (from TCP header)

Returns

Packet destination port

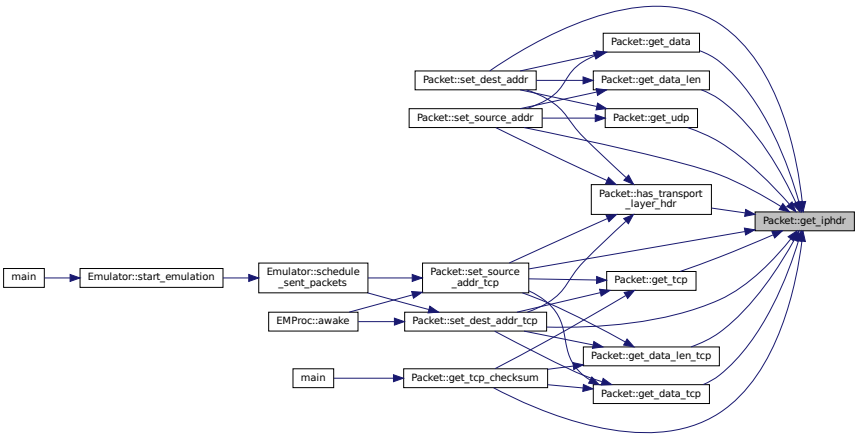
Here is the caller graph for this function:



3.6.3.10 get_iphdr()

```
struct iphdr * Packet::get_iphdr ( ) const  [private]
```

Here is the caller graph for this function:



3.6.3.11 get_size()

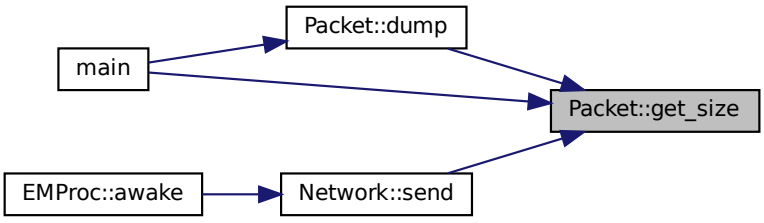
```
size_t Packet::get_size ( ) const
```

Get packet size (private variable)

Returns

Packet size

Here is the caller graph for this function:



3.6.3.12 get_source_addr()

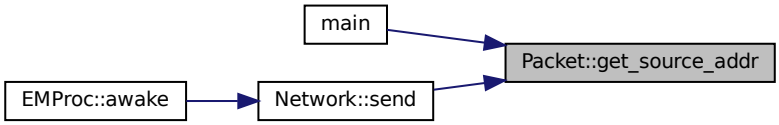
```
std::string Packet::get_source_addr ( ) const
```

Get packet source address (from IPv4 header)

Returns

Packet source address (in number/dot form)

Here is the caller graph for this function:



3.6.3.13 get_source_port()

```
int Packet::get_source_port ( ) const
```

Get packet source port (from UDP header)

Returns

Packet source port

Here is the caller graph for this function:

3.6.3.14 get_source_port_tcp()

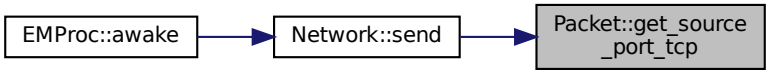
```
int Packet::get_source_port_tcp ( ) const
```

Get packet source port (from TCP header)

Returns

Packet source port

Here is the caller graph for this function:

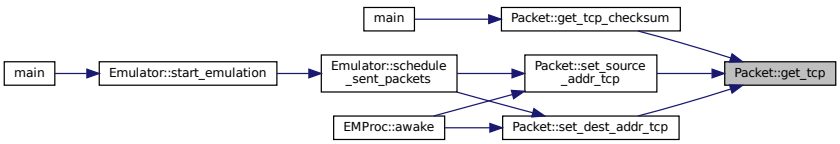


3.6.3.15 get_tcp()

```
struct tcphdr * Packet::get_tcp ( ) const  [private]
```

Here is the call graph for this function:

Here is the caller graph for this function:



3.6.3.16 get_tcp_checksum()

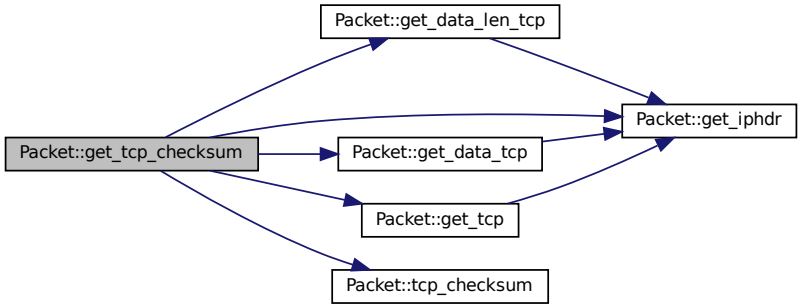
```
int Packet::get_tcp_checksum ( ) const
```

Get packet TCP checksum (from TCP header)

Returns

Packet TCP checksum

Here is the call graph for this function:



Here is the caller graph for this function:

3.6.3.17 get_ts()

```
struct timespec Packet::get_ts ( ) const
```

Get packet timestamp (private variable)

Returns

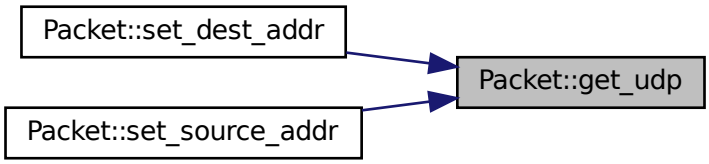
Packet timestamp

3.6.3.18 get_udp()

```
struct udphdr * Packet::get_udp ( ) const  [private]
```

Here is the call graph for this function:

Here is the caller graph for this function:



3.6.3.19 get_version()

```
int Packet::get_version ( ) const
```

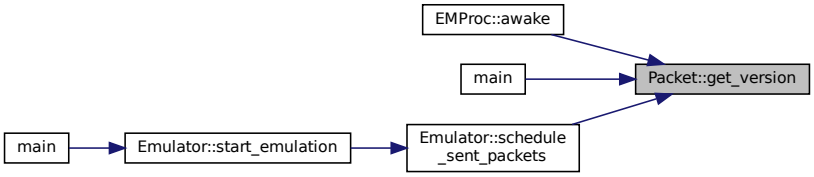
Get packet IPv version (4/6)

Its worth noting that some of packets functionalities don't work for IPv6. Especially setting source or destination addresses are specifically implemented for IPv4

Returns

Packet IPv version (4/6)

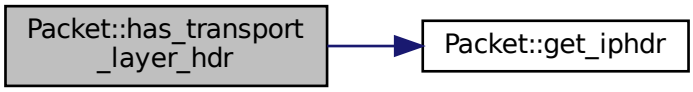
Here is the caller graph for this function:



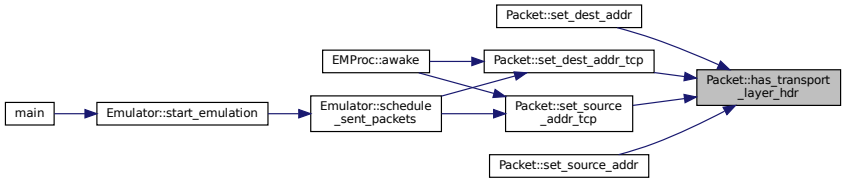
3.6.3.20 has_transport_layer_hdr()

```
bool Packet::has_transport_layer_hdr ( ) const  [private]
```

Here is the call graph for this function:



Here is the caller graph for this function:



3.6.3.21 increase_ts()

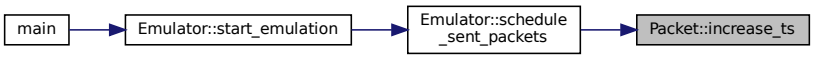
```
void Packet::increase_ts (
            struct timespec other_ts )
```

Increase packet's ts value by `other_ts`.

Parameters

other_ts	Time by which to increate packet's ts value

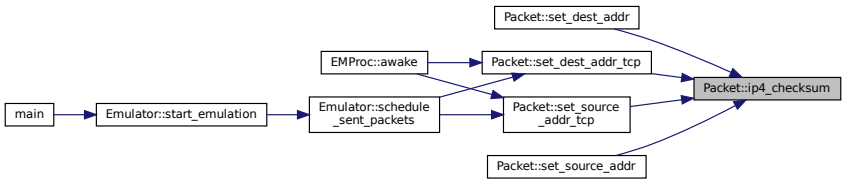
Here is the caller graph for this function:



3.6.3.22 ip4_checksum()

```
uint16_t Packet::ip4_checksum (
            const struct iphdr * ip ) const  [private]
```

Here is the caller graph for this function:



3.6.3.23 operator<()

```
bool Packet::operator< (
            const Packet & other ) const
```

Comparison operator (comparison based on timestamp)

Parameters

other	Packet to compare this on with

3.6.3.24 operator=()

```
Packet & Packet::operator= (
            const Packet & other )
```

Assignment operator.

Parameters

other	Original packet

3.6.3.25 set_dest_addr()

```
void Packet::set_dest_addr (
            const std::string & addr )
```

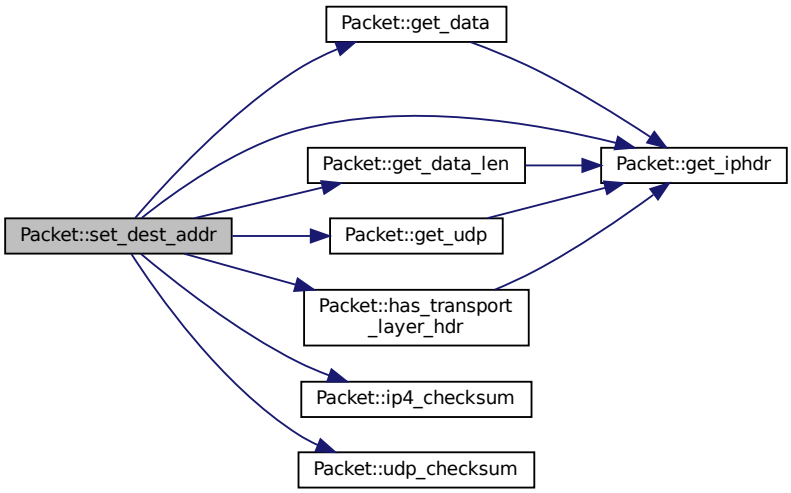
Set a new destination address for the packet.

Changes packets buffer value so that the destination address is updated. This requires the IPv4 checksum to be updated, as well as the UDP checksum in udphdr to be updated.

Parameters

addr	New destination address (in number/dot form)

Here is the call graph for this function:



3.6.3.26 set_dest_addr_tcp()

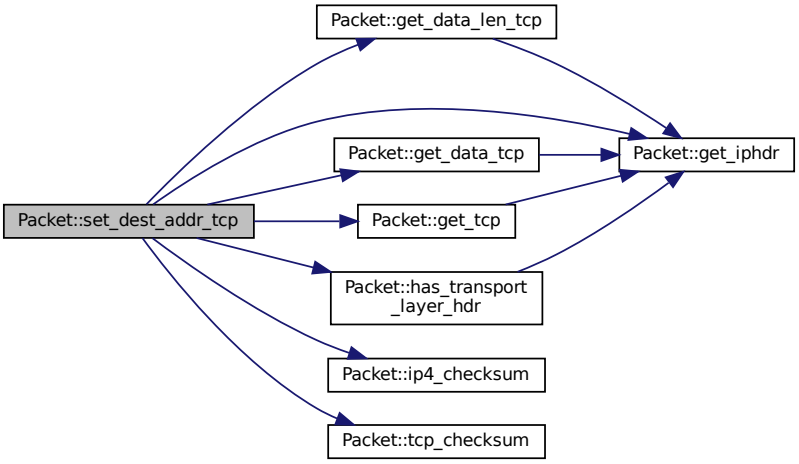
```
void Packet::set_dest_addr_tcp (
            const std::string & addr )
```

Set a new destination address for the packet in TCP.

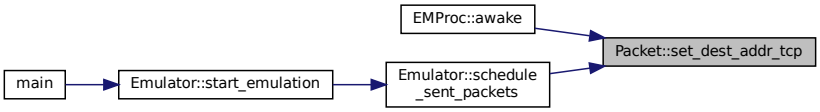
Parameters

addr	New destination address (in number/dot form)

Here is the call graph for this function:



Here is the caller graph for this function:



3.6.3.27 set_source_addr()

```
void Packet::set_source_addr (
            const std::string & addr )
```

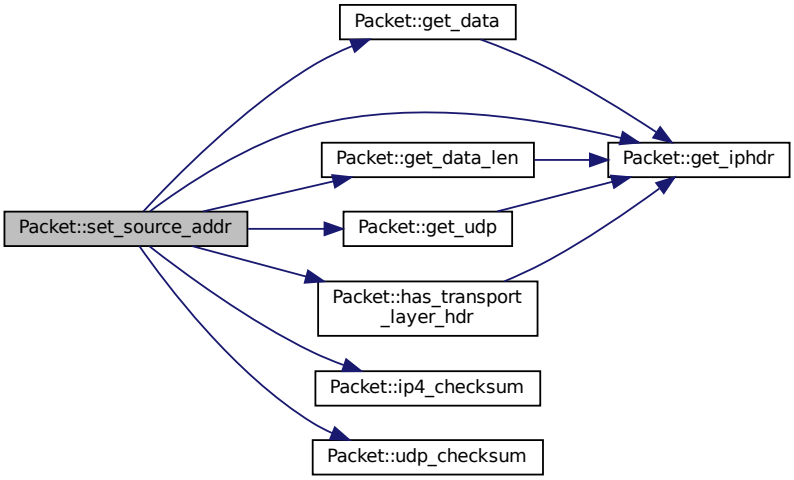
Set a new source address for the packet.

Changes packets buffer value so that the source address is updated. This requires the IPv4 checksum to be updated, as well as the UDP checksum in udphdr to be updated.

Parameters

addr	New source address (in number/dot form)

Here is the call graph for this function:



3.6.3.28 set_source_addr_tcp()

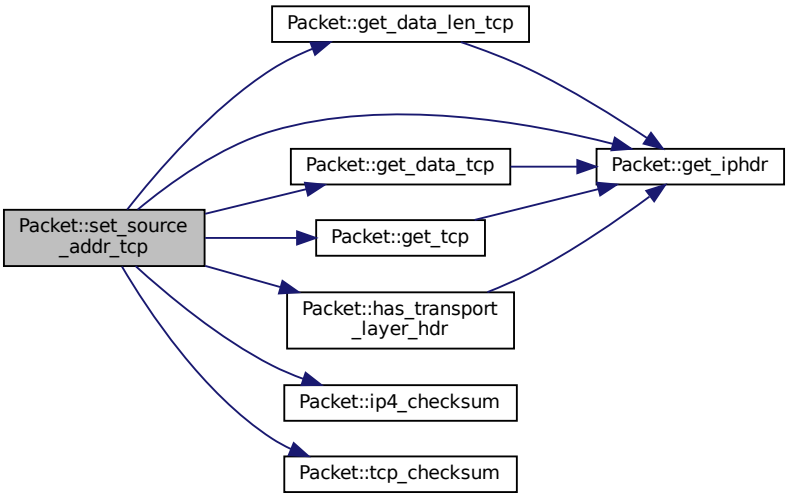
```
void Packet::set_source_addr_tcp (
            const std::string & addr )
```

Set a new source address for the packet in TCP.

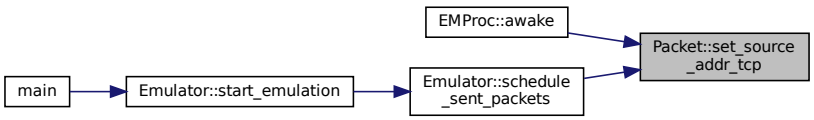
Parameters

addr	New source address (in number/dot form)

Here is the call graph for this function:



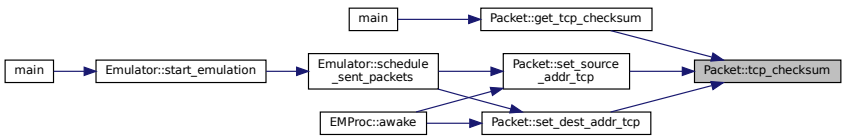
Here is the caller graph for this function:



3.6.3.29 tcp_checksum()

```
uint16_t Packet::tcp_checksum (
            const struct iphdr * ip,
            const struct tcphdr * tcp,
            const char * data,
            size_t data_len ) const  [private]
```

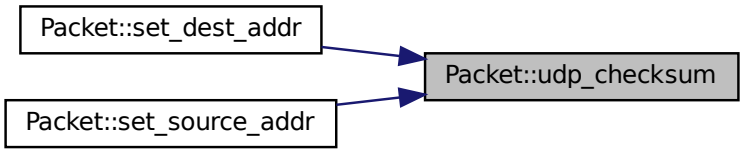
Here is the caller graph for this function:



3.6.3.30 udp_checksum()

```
uint16_t Packet::udp_checksum (
            const struct iphdr * ip,
            const struct udphdr * udp,
            const char * data,
            size_t data_len ) const  [private]
```

Here is the caller graph for this function:



3.6.4 Member Data Documentation

3.6.4.1 buffer

```
char* Packet::buffer  [private]
```

Buffer in which raw data is stored.

3.6.4.2 size

```
size_t Packet::size  [private]
```

Size of data stored in the packet.

3.6.4.3 ts

```
struct timespec Packet::ts  [private]
```

Packets timestamp.

The documentation for this class was generated from the following file:

- packet.hpp

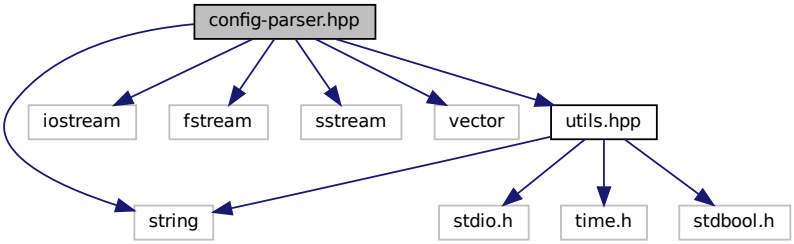
Chapter 4

File Documentation

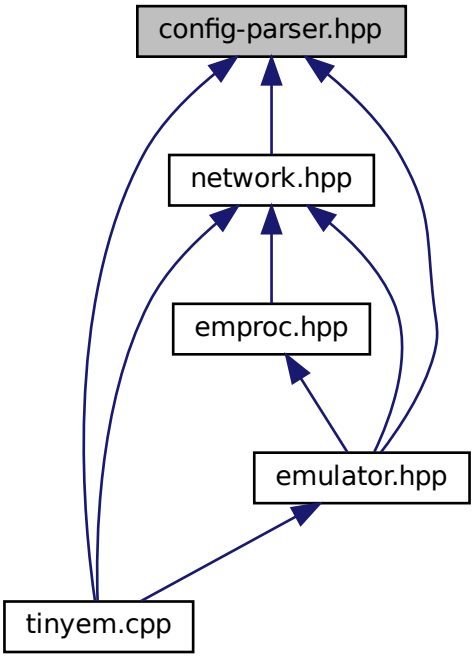
4.1 config-parser.hpp File Reference

```
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include "utils.hpp"
```

Include dependency graph for config-parser.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class ConfigParser

  Class parsing and saving the configuration from a file.

Functions

- std::string CONFIG_PATH ("./configs/config.txt")

  Path to configuration file (default)

Variables

- const int STEPS = 10000

  Number of times emulator awakens some process.

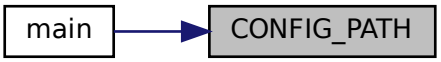
4.1.1 Function Documentation

4.1.1.1 CONFIG_PATH()

```
std::string CONFIG_PATH (
            "./configs/config.txt"  )
```

Path to configuration file (default)

Here is the caller graph for this function:



4.1.2 Variable Documentation

4.1.2.1 STEPS

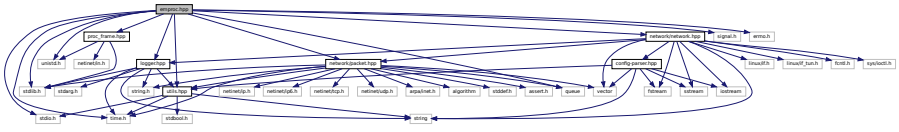
```
const int STEPS = 10000
```

Number of times emulator awakens some process.

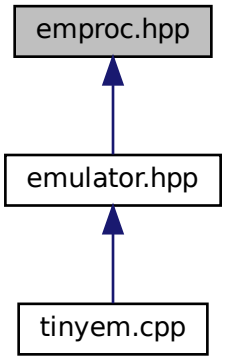
4.2 emproc.hpp File Reference

```
#include <time.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <queue>
#include "proc_frame.hpp"
#include "logger.hpp"
#include "utils.hpp"
#include "network/packet.hpp"
#include "network/network.hpp"
```

Include dependency graph for emproc.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class EMProc

  Controller of a single process inside an emulation.

Typedefs

- typedef int em_id_t

4.2.1 Typedef Documentation

4.2.1.1 em_id_t

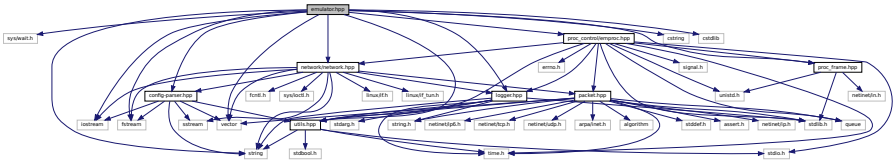
```
typedef int em_id_t
```

Type of emulator's internal process id

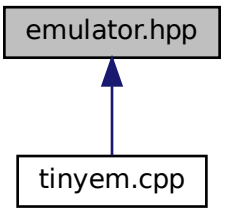
4.3 emulator.hpp File Reference

```
#include <sys/wait.h>
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <cstdlib>
#include "utils.hpp"
#include "logger.hpp"
#include "config-parser.hpp"
#include "network/network.hpp"
#include "proc_control/emproc.hpp"
#include "proc_control/proc_frame.hpp"
```

Include dependency graph for emulator.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Emulator

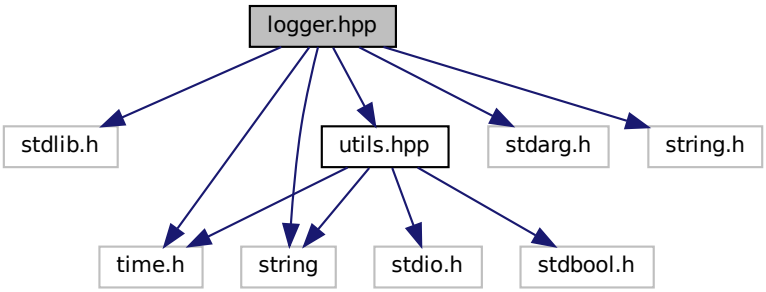
  Class encapsulating the main logic of the emulator.

4.4 logger.hpp File Reference

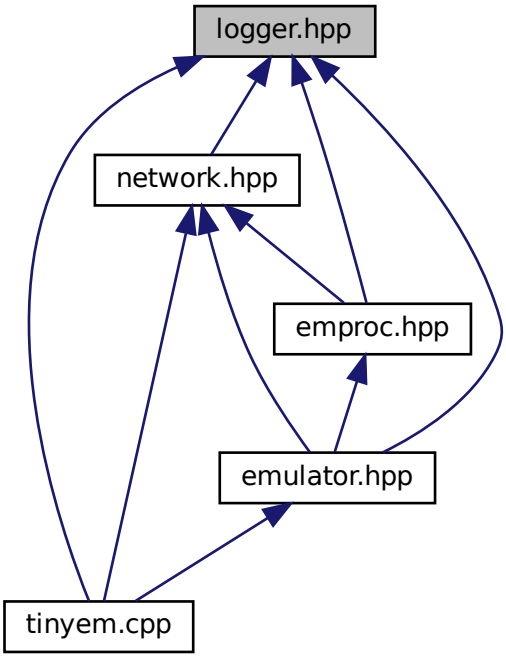
```
#include <stdlib.h>
#include <time.h>
#include <stdarg.h>
```

```
#include <string.h>
#include <string>
#include "utils.hpp"
```

Include dependency graph for logger.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Logger

  Utility class for logging in the system.

Variables

- Logger * logger_ptr

  Pointer to global logger object.

4.4.1 Variable Documentation

4.4.1.1 logger_ptr

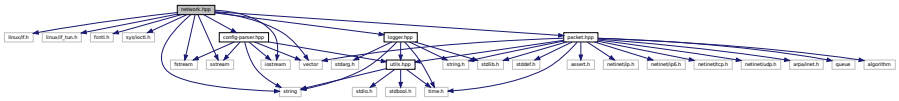
```
Logger* logger_ptr  [extern]
```

Pointer to global logger object.

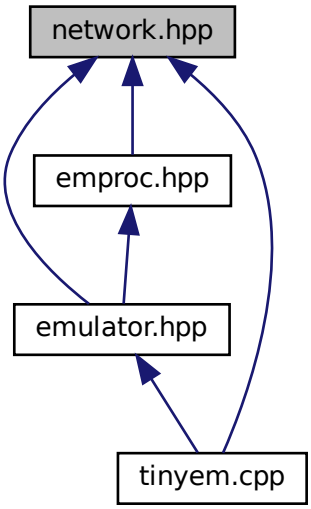
4.5 network.hpp File Reference

```
#include <linux/if.h>
#include <linux/if_tun.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <vector>
#include <fstream>
#include <sstream>
#include <string>
#include <iostream>
#include "packet.hpp"
#include "config-parser.hpp"
#include "logger.hpp"
```

Include dependency graph for network.hpp:



This graph shows which files directly or indirectly include this file:



Classes

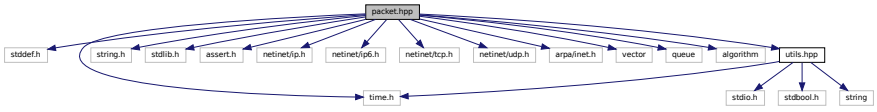
- class Network

  Class responsible for all network control and communication.

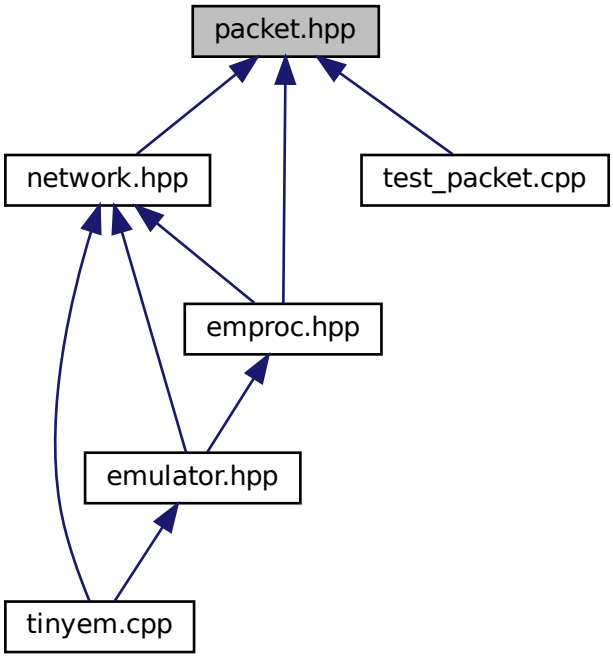
4.6 packet.hpp File Reference

```
#include <stddef.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <netinet/ip.h>
#include <netinet/ip6.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <arpa/inet.h>
#include <vector>
#include <queue>
#include <algorithm>
#include "utils.hpp"
```

Include dependency graph for packet.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Packet

  Class encapsulating single ip frame sent through TUN interface.

Macros

- #define _DEFAULT_SOURCE 1
- #define MTU 1500

4.6.1 Macro Definition Documentation

4.6.1.1 _DEFAULT_SOURCE

```
#define _DEFAULT_SOURCE 1
```

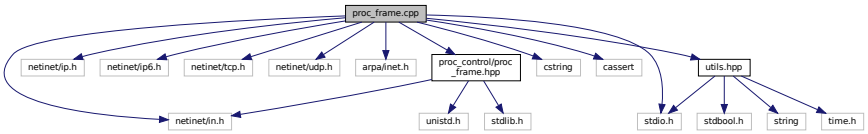
4.6.1.2 MTU

```
#define MTU 1500
```

4.7 proc_frame.cpp File Reference

```
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/ip6.h>
#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <cstring>
#include <cassert>
#include "proc_control/proc_frame.hpp"
#include "utils.hpp"
```

Include dependency graph for proc_frame.cpp:



Macros

- #define _DEFAULT_SOURCE 1

Functions

- static void write_or_die (const char *buf, size_t len)
- void dump (const char *buf, size_t len)

  Prints the buffer in hex.
- void dump_short (const char *buf, size_t len)
- static uint16_t ip4_checksum (const struct iphdr *ip)
- void process (const char *buf, size_t len)
- void process_ip6 (const char *buf, size_t len)
- static void process_ip4_tcp_payload (const struct iphdr *ip __attribute__((unused)), const struct tcphdr *tcp __attribute__((unused)), const char *buf, size_t len)
- void process_ip4_tcp (const struct iphdr *ip, const char *buf, size_t len)
- void process_ip4_udp (const struct iphdr *ip, const char *buf, size_t len)
- void process_ip4 (const char *buf, size_t len)

4.7.1 Macro Definition Documentation

4.7.1.1 _DEFAULT_SOURCE

```
#define _DEFAULT_SOURCE 1
```

4.7.2 Function Documentation

4.7.2.1 dump()

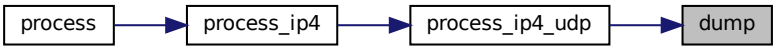
```
void dump (
            const char * buf,
            size_t len )
```

Prints the buffer in hex.

Parameters

buf	The buffer to print
len	The length of the buffer

Here is the caller graph for this function:



4.7.2.2 dump_short()

```
void dump_short (
            const char * buf,
            size_t len )
```

Here is the caller graph for this function:

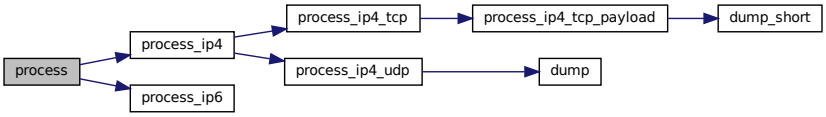
4.7.2.3 ip4_checksum()

```
static uint16_t ip4_checksum (
            const struct iphdr * ip )  [static]
```

4.7.2.4 process()

```
void process (
            const char * buf,
            size_t len )
```

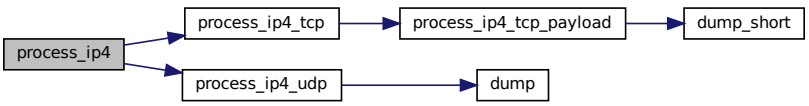
Here is the call graph for this function:



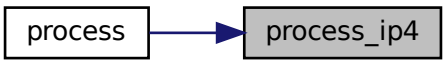
4.7.2.5 process_ip4()

```
void process_ip4 (
            const char * buf,
            size_t len )
```

Here is the call graph for this function:



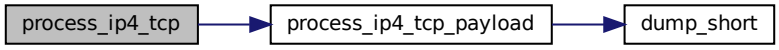
Here is the caller graph for this function:



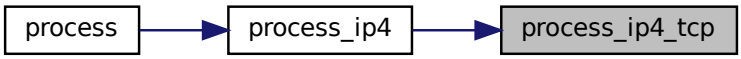
4.7.2.6 process_ip4_tcp()

```
void process_ip4_tcp (
            const struct iphdr * ip,
            const char * buf,
            size_t len )
```

Here is the call graph for this function:



Here is the caller graph for this function:

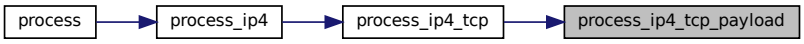


4.7.2.7 process_ip4_tcp_payload()

```
static void process_ip4_tcp_payload (
            const struct iphdr *ip  __attribute__(unused),
            const struct tcphdr *tcp  __attribute__(unused),
            const char * buf,
            size_t len )  [static]
```

Here is the call graph for this function:

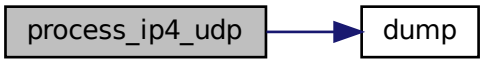
Here is the caller graph for this function:



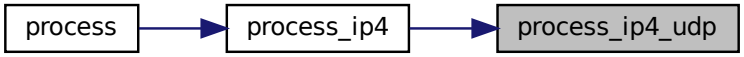
4.7.2.8 process_ip4_udp()

```
void process_ip4_udp (
            const struct iphdr * ip,
            const char * buf,
            size_t len )
```

Here is the call graph for this function:



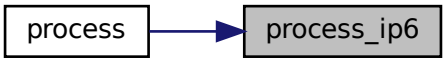
Here is the caller graph for this function:



4.7.2.9 process_ip6()

```
void process_ip6 (
            const char * buf,
            size_t len )
```

Here is the caller graph for this function:



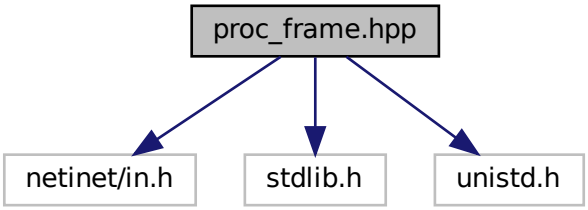
4.7.2.10 write_or_die()

```
static void write_or_die (
            const char * buf,
            size_t len )  [static]
```

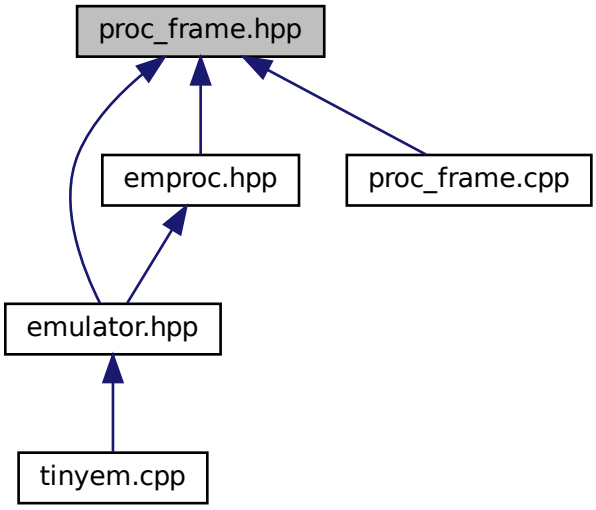
4.8 proc_frame.hpp File Reference

```
#include <netinet/in.h>
#include <stdlib.h>
#include <unistd.h>
```

Include dependency graph for proc_frame.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- void dump (const char *buf, size_t len)

  Prints the buffer in hex.
- void process (const char *buf, size_t len)
- void process_ip6 (const char *buf, size_t len)
- void process_ip4 (const char *buf, size_t len)
- void process_ip4_tcp (const struct iphdr *ip, const char *buf, size_t len)
- void process_ip4_udp (const struct iphdr *ip, const char *buf, size_t len)

4.8.1 Function Documentation

4.8.1.1 dump()

```
void dump (
            const char * buf,
            size_t len )
```

Prints the buffer in hex.

Parameters

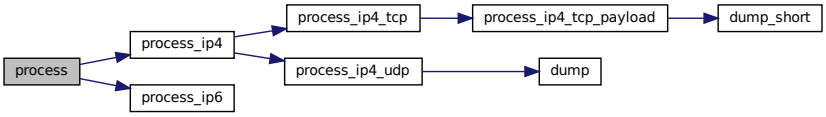
buf	The buffer to print
len	The length of the buffer

Here is the caller graph for this function:

4.8.1.2 process()

```
void process (
            const char * buf,
            size_t len )
```

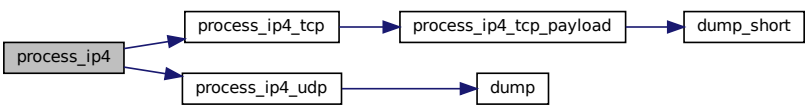
Here is the call graph for this function:



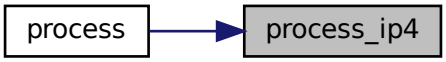
4.8.1.3 process_ip4()

```
void process_ip4 (
            const char * buf,
            size_t len )
```

Here is the call graph for this function:



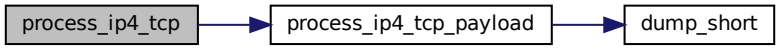
Here is the caller graph for this function:



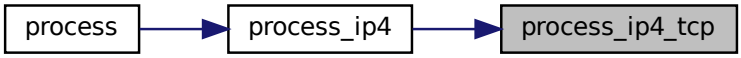
4.8.1.4 process_ip4_tcp()

```
void process_ip4_tcp (
            const struct iphdr * ip,
            const char * buf,
            size_t len )
```

Here is the call graph for this function:



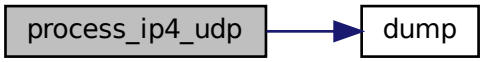
Here is the caller graph for this function:



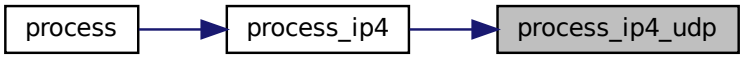
4.8.1.5 process_ip4_udp()

```
void process_ip4_udp (
            const struct iphdr * ip,
            const char * buf,
            size_t len )
```

Here is the call graph for this function:



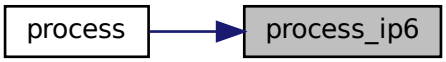
Here is the caller graph for this function:



4.8.1.6 process_ip6()

```
void process_ip6 (
            const char * buf,
            size_t len )
```

Here is the caller graph for this function:

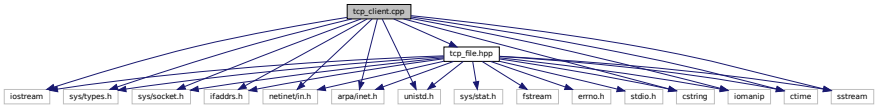


4.9 tcp_client.cpp File Reference

```
#include <iostream>
#include <sys/types.h>
#include <sys/socket.h>
#include <ifaddrs.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#include <iomanip>
#include <ctime>
#include <sstream>
#include "tcp_file.hpp"
```

Include dependency graph for tcp_client.cpp:



Functions

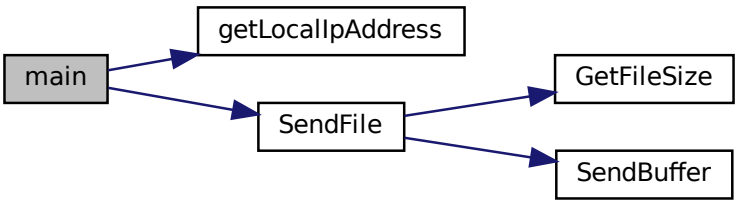
- int main (int argc, char ∗argv[ ])

4.9.1 Function Documentation

4.9.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

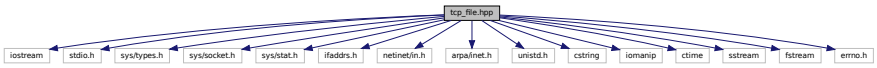
Here is the call graph for this function:



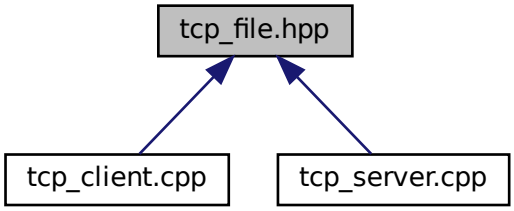
4.10 tcp_file.hpp File Reference

```
#include <iostream>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <ifaddrs.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring>
#include <iomanip>
#include <ctime>
#include <sstream>
#include <fstream>
#include <errno.h>
```

Include dependency graph for tcp_file.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- void dump (const char *buf, size_t len)

  Prints the buffer in hex.
- std::string getLocalIpAddress ()

  Get the Local Ip Address object.
- std::string getLocalTime ()

  Get the Local Time object.
- int64_t GetFileSize (const std::string &fileName)

  Get the File Size object.
- int RecvBuffer (int socketFd, char *buffer, int bufferSize, int chunkSize=16 *1024)

  Recieves data in to buffer until bufferSize value is met.

- int SendBuffer (int socketFd, const char *buffer, int bufferSize, int chunkSize=64 *1024)

  Sends data in buffer until bufferSize value is met, return size sent.
- int64_t SendFile (int socketFd, const std::string &fileName, int chunkSize=32 *1024)

  Sends a file.
- int64_t RecvFile (int socketFd, const std::string &fileName, int chunkSize=64 *1024)

  Receives a file.

4.10.1 Function Documentation

4.10.1.1 dump()

```
void dump (
            const char * buf,
            size_t len )
```

Prints the buffer in hex.

Parameters

buf	The buffer to print
len	The length of the buffer

4.10.1.2 GetFileSize()

```
int64_t GetFileSize (
            const std::string & fileName )
```

Get the File Size object.

Parameters

fileName	The filename to get the size of

Returns

The size of the file

Reference for this function Here is the caller graph for this function:

4.10.1.3 getLocalIpAddress()

```
std::string getLocalIpAddress ( )
```

Get the Local Ip Address object.

Returns

The local IP address

Here is the caller graph for this function:

4.10.1.4 getLocalTime()

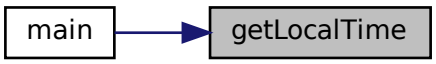
```
std::string getLocalTime ( )
```

Get the Local Time object.

Returns

The local time

Here is the caller graph for this function:



4.10.1.5 RecvBuffer()

```
int RecvBuffer (
            int socketFd,
            char * buffer,
            int bufferSize,
            int chunkSize = 16 * 1024 )
```

Recieves data in to buffer until bufferSize value is met.

Parameters

socketFd	The receiver socket file descriptor
buffer	The buffer to receive data into
bufferSize	The size of the buffer
chunkSize	The size of the chunk to receive at a time

Returns

The size of the buffer received

Here is the caller graph for this function:

4.10.1.6 RecvFile()

```
int64_t RecvFile (
            int socketFd,
            const std::string & fileName,
            int chunkSize = 64 * 1024 )
```

Receives a file.

returns size of file if success
returns -1 if file couldn't be opened for output
returns -2 if couldn't receive file length properly
returns -3 if couldn't receive file properly

Parameters

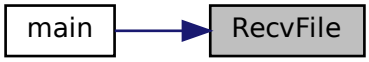
socketFd	The receiver socket file descriptor
fileName	The filename to receive
chunkSize	The size of the chunk to receive at a time

Returns

The size of the file received

Here is the call graph for this function:

Here is the caller graph for this function:



4.10.1.7 SendBuffer()

```
int SendBuffer (
            int socketFd,
            const char * buffer,
            int bufferSize,
            int chunkSize = 64 * 1024 )
```

Sends data in buffer until bufferSize value is met, return size sent.

Parameters

socketFd	The sender socket file descriptor
buffer	The buffer to send data from
bufferSize	The size of the buffer
chunkSize	The size of the chunk to send at a time

Returns

The size of the buffer sent

Here is the caller graph for this function:

4.10.1.8 SendFile()

```
int64_t SendFile (
            int socketFd,
            const std::string & fileName,
            int chunkSize = 32 * 1024 )
```

Sends a file.

returns size of file if success
returns -1 if file couldn't be opened for input
returns -2 if couldn't send file length properly
returns -3 if file couldn't be sent properly

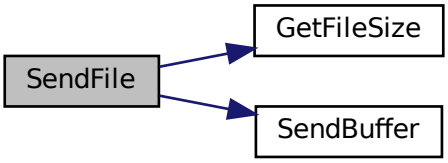
Parameters

socketFd	The sender socket file descriptor
fileName	The filename to send
chunkSize	The size of the chunk to send at a time

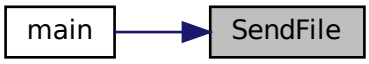
Returns

The size of the file sent

Here is the call graph for this function:



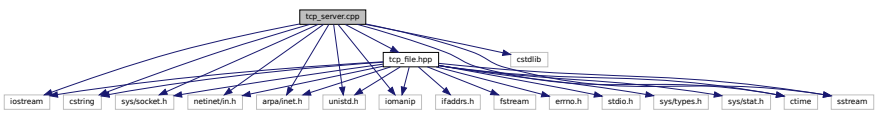
Here is the caller graph for this function:



4.11 tcp_server.cpp File Reference

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <iomanip>
#include <ctime>
#include <sstream>
#include "tcp_file.hpp"
```

Include dependency graph for tcp_server.cpp:



Functions

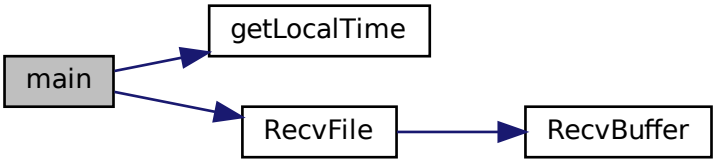
- int main (int argc, char ∗argv[ ])

4.11.1 Function Documentation

4.11.1.1 main()

```
int main (
            int argc,
            char * argv[] )
```

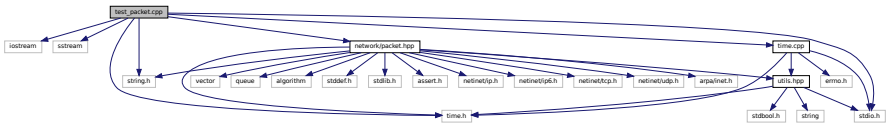
Here is the call graph for this function:



4.12 test_packet.cpp File Reference

```
#include <iostream>
#include <sstream>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "network/packet.hpp"
#include "time.cpp"
```

Include dependency graph for test_packet.cpp:



Functions

- unsigned char hexCharToByte (unsigned char c)
- void hexStringToCharArray (const std::string &hexString, char ∗buffer)
- int main ()

4.12.1 Function Documentation

4.12.1.1 hexCharToByte()

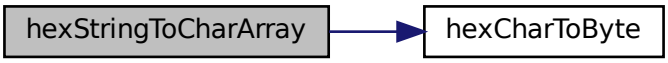
```
unsigned char hexCharToByte (
            unsigned char c )
```

Here is the caller graph for this function:

4.12.1.2 hexStringToCharArray()

```
void hexStringToCharArray (
            const std::string & hexString,
            char * buffer )
```

Here is the call graph for this function:

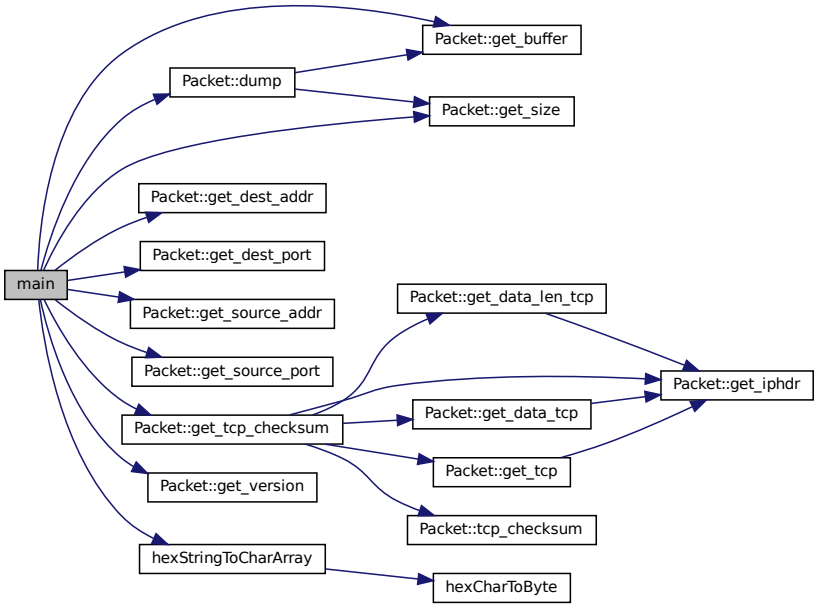


Here is the caller graph for this function:

4.12.1.3 main()

```
int main ( )
```

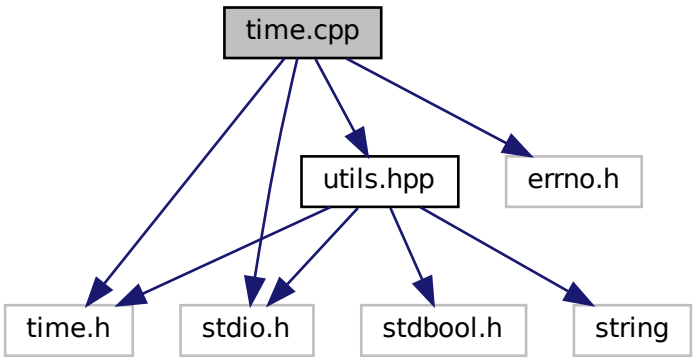
Here is the call graph for this function:



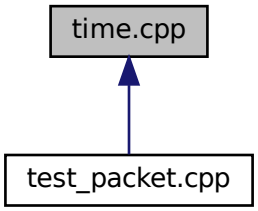
4.13 time.cpp File Reference

```
#include <time.h>
#include <errno.h>
#include <stdio.h>
#include "utils.hpp"
```

Include dependency graph for time.cpp:



This graph shows which files directly or indirectly include this file:



Functions

- void real_sleep (long long nsecs)
- long long nano_from_ts (struct timespec ts)
- struct timespec ts_from_nano (long long nsecs)
- struct timespec operator+ (const struct timespec &ts1, const struct timespec &ts2)
- struct timespec operator- (const struct timespec &ts1, const struct timespec &ts2)
- struct timespec operator∗ (double x, const struct timespec &ts)
- struct timespec operator∗ (const struct timespec &ts, double x)
- bool operator> (const struct timespec &ts1, const struct timespec &ts2)
- bool operator< (const struct timespec &ts1, const struct timespec &ts2)
- bool check_if_elapsed (struct timespec ts1, struct timespec ts2)
- struct timespec get_time_since (struct timespec ts1)
- int get_sec (struct timespec t)
- int get_msec (struct timespec t)
- int get_micsec (struct timespec t)
- int get_nsec (struct timespec t)

4.13.1 Function Documentation

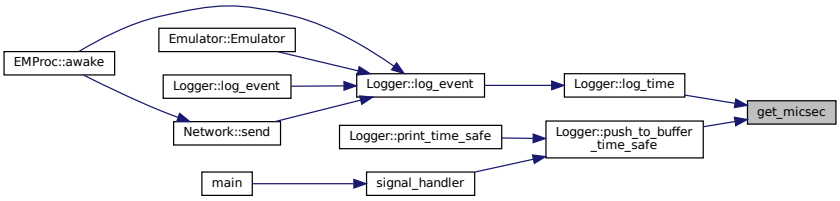
4.13.1.1 check_if_elapsed()

```
bool check_if_elapsed (
            struct timespec ts1,
            struct timespec ts2 )
```

4.13.1.2 get_micsec()

```
int get_micsec (
            struct timespec t )
```

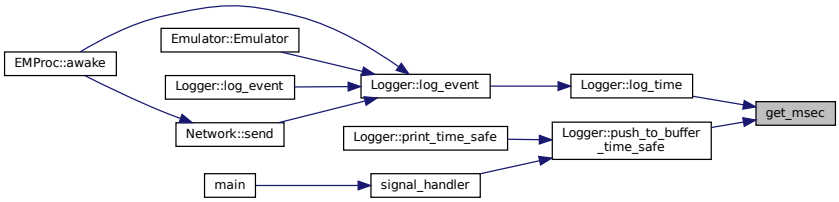
Here is the caller graph for this function:



4.13.1.3 get_msec()

```
int get_msec (
            struct timespec t )
```

Here is the caller graph for this function:



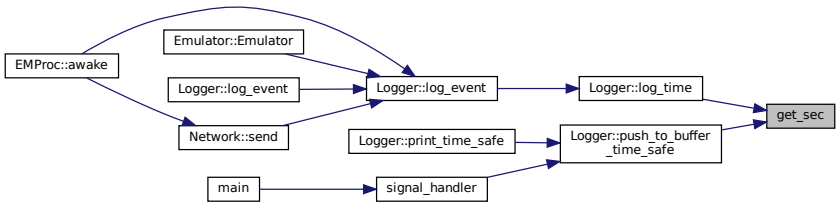
4.13.1.4 get_nsec()

```
int get_nsec (
            struct timespec t )
```

4.13.1.5 get_sec()

```
int get_sec (
            struct timespec t )
```

Here is the caller graph for this function:



4.13.1.6 get_time_since()

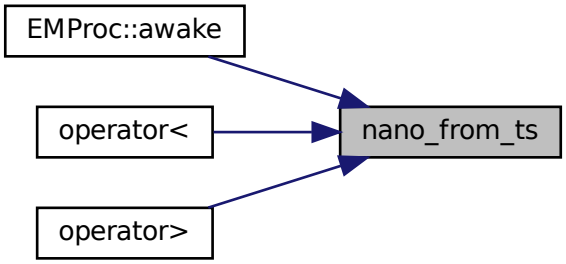
```
struct timespec get_time_since (
            struct timespec ts1 )
```

Here is the caller graph for this function:

4.13.1.7 nano_from_ts()

```
long long nano_from_ts (
            struct timespec ts )
```

Here is the caller graph for this function:



4.13.1.8 operator*() [1/2]

```
struct timespec operator* (
            const struct timespec & ts,
            double x )
```

4.13.1.9 operator*() [2/2]

```
struct timespec operator* (
            double x,
            const struct timespec & ts )
```

4.13.1.10 operator+()

```
struct timespec operator+ (
            const struct timespec & ts1,
            const struct timespec & ts2 )
```

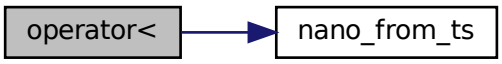
4.13.1.11 operator-()

```
struct timespec operator- (
            const struct timespec & ts1,
            const struct timespec & ts2 )
```

4.13.1.12 operator<()

```
bool operator< (
            const struct timespec & ts1,
            const struct timespec & ts2 )
```

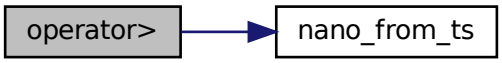
Here is the call graph for this function:



4.13.1.13 operator>()

```
bool operator> (
            const struct timespec & ts1,
            const struct timespec & ts2 )
```

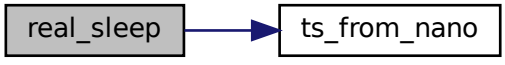
Here is the call graph for this function:



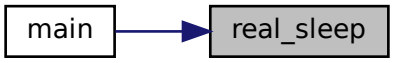
4.13.1.14 real_sleep()

```
void real_sleep (
            long long nsecs )
```

Here is the call graph for this function:



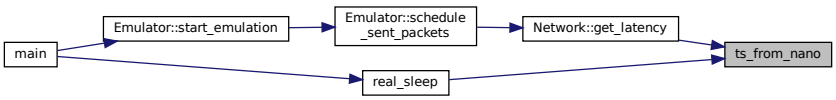
Here is the caller graph for this function:



4.13.1.15 ts_from_nano()

```
struct timespec ts_from_nano (
            long long nsecs )
```

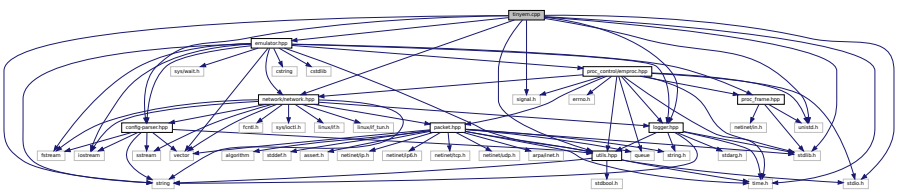
Here is the caller graph for this function:



4.14 tinyem.cpp File Reference

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <string>
#include "config-parser.hpp"
#include "utils.hpp"
#include "logger.hpp"
#include "network/network.hpp"
#include "emulator.hpp"
```

Include dependency graph for tinyem.cpp:



Functions

- void signal_handler (int signum)
- int main (int argc, const char **argv)

Variables

- Logger * logger_ptr = nullptr

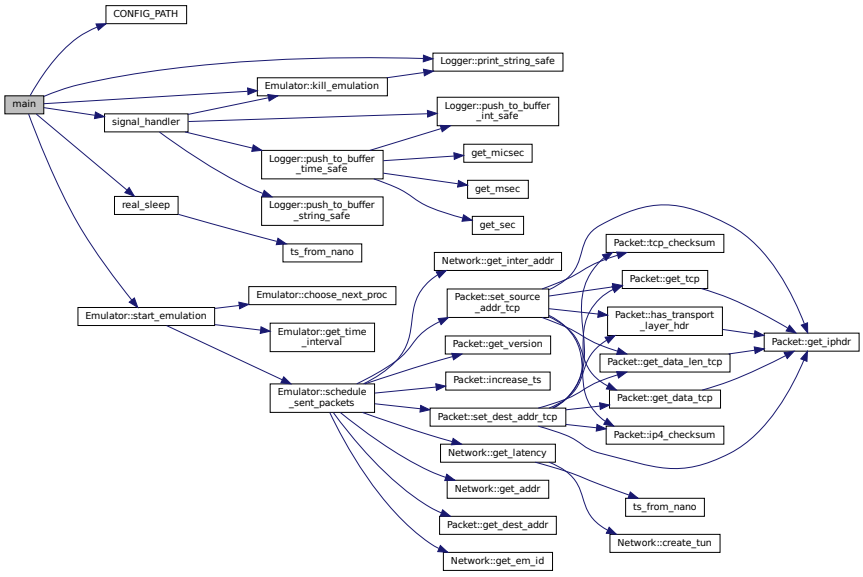
  Pointer to global logger object.
- Emulator * em_ptr = nullptr

4.14.1 Function Documentation

4.14.1.1 main()

```
int main (
            int argc,
            const char ** argv )
```

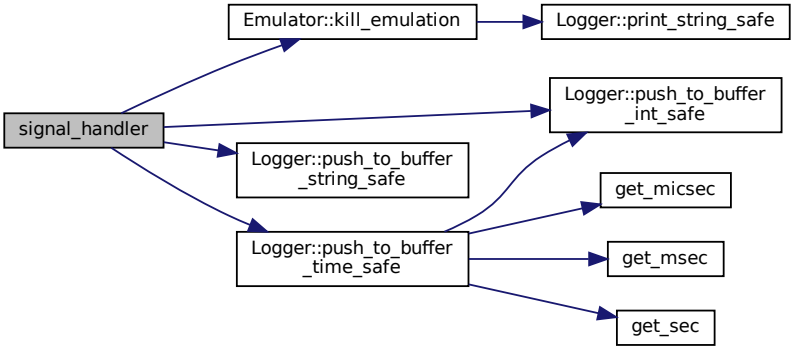
Here is the call graph for this function:



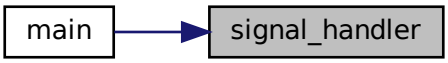
4.14.1.2 signal_handler()

```
void signal_handler (
            int signum )
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.14.2 Variable Documentation

4.14.2.1 em_ptr

```
Emulator* em_ptr = nullptr
```

4.14.2.2 logger_ptr

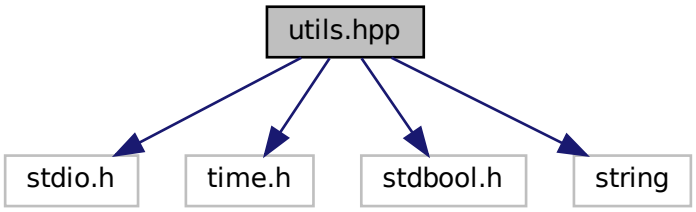
```
Logger* logger_ptr = nullptr
```

Pointer to global logger object.

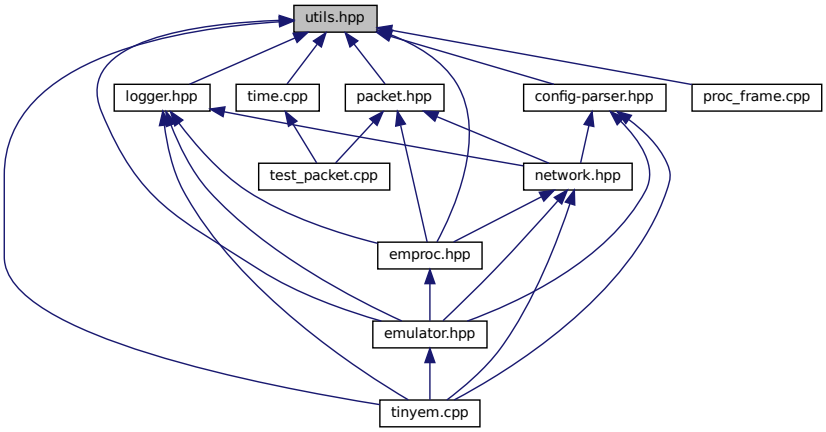
4.15 utils.hpp File Reference

```
#include <stdio.h>
#include <time.h>
#include <stdbool.h>
#include <string>
```

Include dependency graph for utils.hpp:



This graph shows which files directly or indirectly include this file:



Macros

- #define NANOSECOND 1
- #define MICROSECOND (int)1e3
- #define MILLISECOND (int)1e6
- #define SECOND (int)1e9
- #define BUF_SIZE 320
- #define panic(_str) do { perror(_str); abort(); } while (0)

Functions

- void real_sleep (long long nsecs)
- long long nano_from_ts (struct timespec ts)
- struct timespec ts_from_nano (long long nsecs)
- bool check_if_elapsed (struct timespec ts1, struct timespec ts2)
- struct timespec get_time_since (struct timespec ts1)
- struct timespec operator+ (const struct timespec &ts1, const struct timespec &ts2)
- struct timespec operator- (const struct timespec &ts1, const struct timespec &ts2)
- struct timespec operator∗ (double x, const struct timespec &ts)
- struct timespec operator∗ (const struct timespec &ts, double x)
- bool operator> (const struct timespec &ts1, const struct timespec &ts2)
- bool operator< (const struct timespec &ts1, const struct timespec &ts2)
- int get_sec (struct timespec t)
- int get_msec (struct timespec t)
- int get_micsec (struct timespec t)
- int get_nsec (struct timespec t)

4.15.1 Macro Definition Documentation

4.15.1.1 BUF_SIZE

```
#define BUF_SIZE 320
```

4.15.1.2 MICROSECOND

```
#define MICROSECOND (int)1e3
```

4.15.1.3 MILLISECOND

```
#define MILLISECOND (int)1e6
```

4.15.1.4 NANOSECOND

```
#define NANOSECOND 1
```

4.15.1.5 panic

```
#define panic(
             _str ) do { perror(_str); abort(); } while (0)
```

4.15.1.6 SECOND

```
#define SECOND (int)1e9
```

4.15.2 Function Documentation

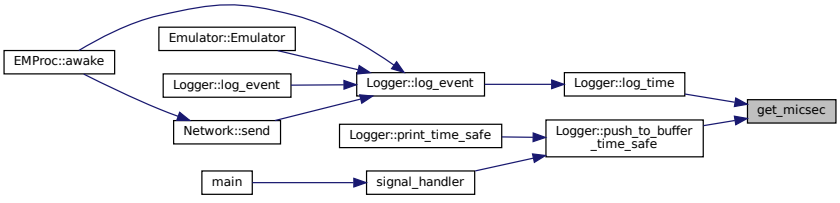
4.15.2.1 check_if_elapsed()

```
bool check_if_elapsed (
             struct timespec ts1,
             struct timespec ts2 )
```

4.15.2.2 get_micsec()

```
int get_micsec (
             struct timespec t )
```

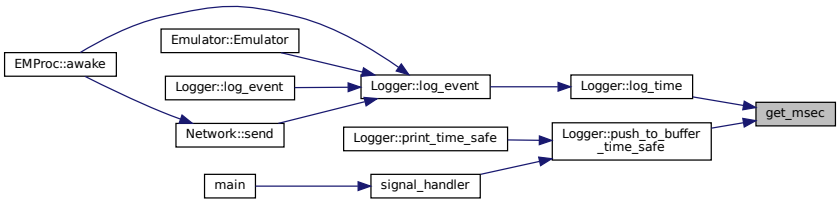
Here is the caller graph for this function:



4.15.2.3 get_msec()

```
int get_msec (
            struct timespec t )
```

Here is the caller graph for this function:



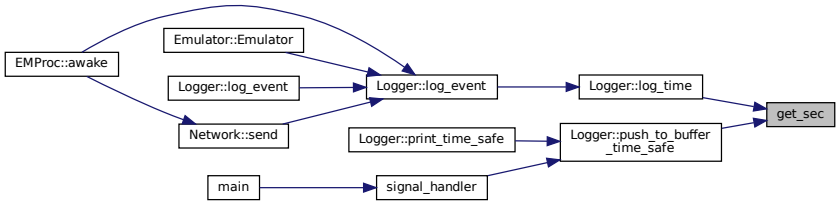
4.15.2.4 get_nsec()

```
int get_nsec (
            struct timespec t )
```

4.15.2.5 get_sec()

```
int get_sec (
            struct timespec t )
```

Here is the caller graph for this function:



4.15.2.6 get_time_since()

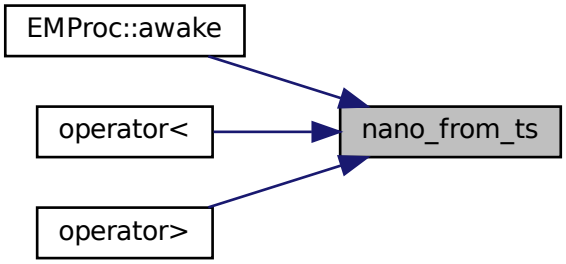
```
struct timespec get_time_since (
            struct timespec ts1 )
```

Here is the caller graph for this function:

4.15.2.7 nano_from_ts()

```
long long nano_from_ts (
            struct timespec ts )
```

Here is the caller graph for this function:



4.15.2.8 operator*() [1/2]

```
struct timespec operator* (
            const struct timespec & ts,
            double x )
```

4.15.2.9 operator*() [2/2]

```
struct timespec operator* (
            double x,
            const struct timespec & ts )
```

4.15.2.10 operator+()

```
struct timespec operator+ (
            const struct timespec & ts1,
            const struct timespec & ts2 )
```

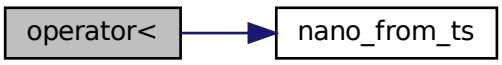
4.15.2.11 operator-()

```
struct timespec operator- (
            const struct timespec & ts1,
            const struct timespec & ts2 )
```

4.15.2.12 operator<()

```
bool operator< (
            const struct timespec & ts1,
            const struct timespec & ts2 )
```

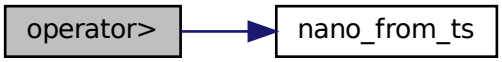
Here is the call graph for this function:



4.15.2.13 operator>()

```
bool operator> (
            const struct timespec & ts1,
            const struct timespec & ts2 )
```

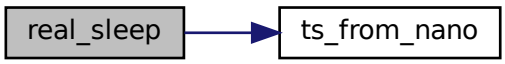
Here is the call graph for this function:



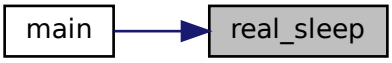
4.15.2.14 real_sleep()

```
void real_sleep (
            long long nsecs )
```

Here is the call graph for this function:



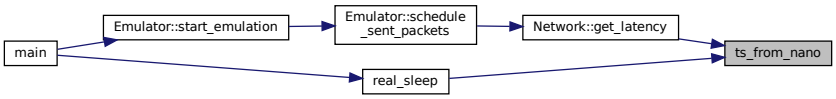
Here is the caller graph for this function:



4.15.2.15 ts_from_nano()

```
struct timespec ts_from_nano (
            long long nsecs )
```

Here is the caller graph for this function:



Index

SimpleEM Documentation

Dummy Part

Generated by Doxygen 1.9.1

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

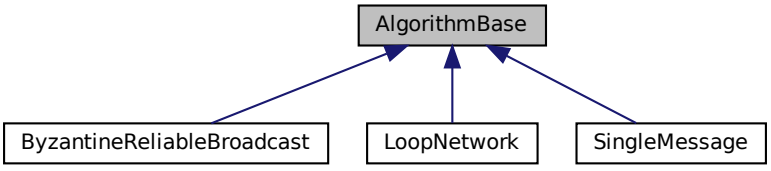
Class Documentation

3.1 AlgorithmBase Class Reference

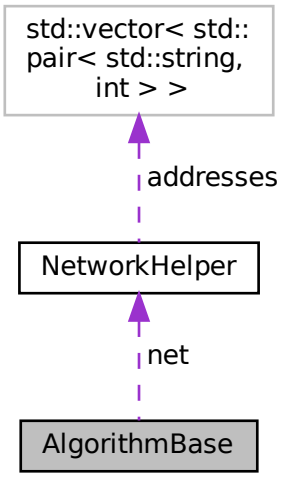
Base class for all distributed network / algorithm testing.

`#include <algorithm-base.hpp>`

Inheritance diagram for AlgorithmBase:



Collaboration diagram for AlgorithmBase:



Public Member Functions

- **AlgorithmBase** (int em_id, NetworkHelper &net)
- void **broadcast** (int em_id, const std::string &message)
- message_t **force_receive** ()

Protected Member Functions

- std::string **get_mes_type** (const std::string &message) const
- std::string **get_mes_value** (const std::string &message) const

Protected Attributes

- int **em_id**
- NetworkHelper & **net**

3.1.1 Detailed Description

Base class for all distributed network / algorithm testing.

This class provides some basic functions for all algorithms, such as broadcast and receive.

The documentation for this class was generated from the following file:

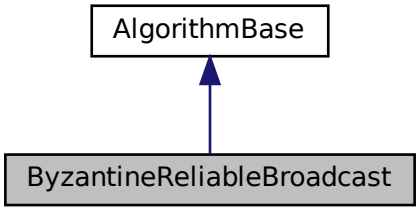
- include/algorithms/algorithm-base.hpp

3.2 ByzantineReliableBroadcast Class Reference

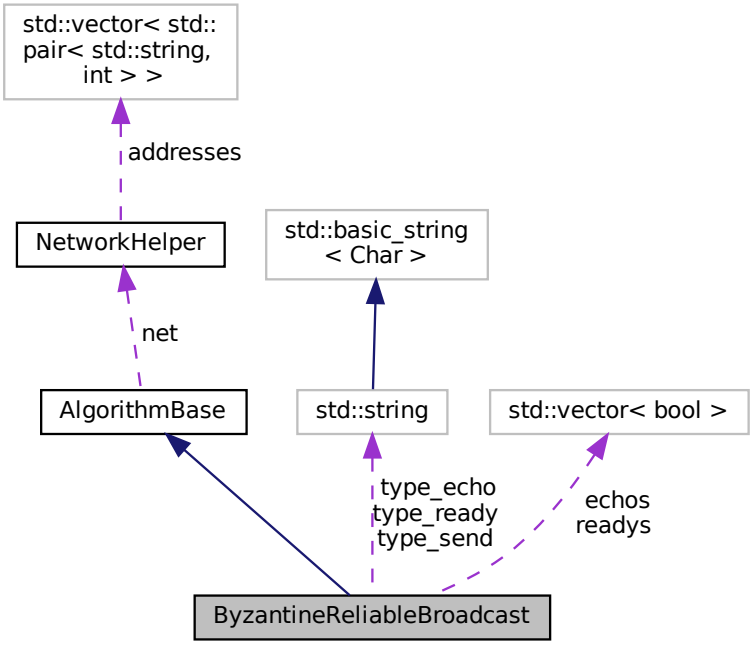
For byzantine reliable broadcast test.

`#include <byzantine-reliable-broadcast.hpp>`

Inheritance diagram for ByzantineReliableBroadcast:



Collaboration diagram for ByzantineReliableBroadcast:



Public Member Functions

- void start (const std::string &message)

  0 broadcasts, but is also part of senders
- **AlgorithmBase** (int em_id, NetworkHelper &net)

Private Attributes

- const std::string **type_send** = std::string("SEND")
- const std::string **type_echo** = std::string("ECHO")
- const std::string **type_ready** = std::string("READY")
- bool **sentecho**
- bool **sentready**
- bool **delivered**
- std::vector< bool > **echos**
- std::vector< bool > **readys**

Additional Inherited Members

3.2.1 Detailed Description

For byzantine reliable broadcast test.

The documentation for this class was generated from the following file:

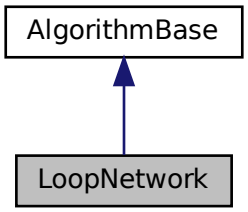
- include/algorithms/byzantine-reliable-broadcast.hpp

3.3 LoopNetwork Class Reference

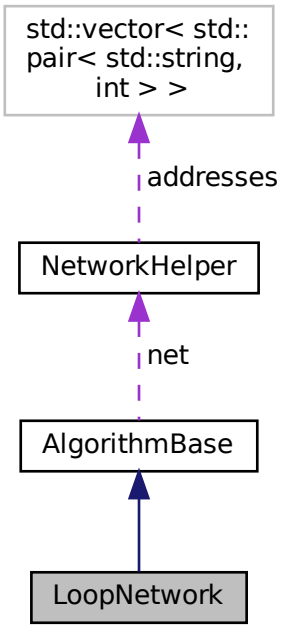
For loop network test.

`#include <loop-network.hpp>`

Inheritance diagram for LoopNetwork:



Collaboration diagram for LoopNetwork:



Public Member Functions

- void start (const std::string &message, int loops)

  Every node passes the message to the next node after receiving it. In total `loops` are done. 0 starts the loop.
- **AlgorithmBase** (int em_id, NetworkHelper &net)

Additional Inherited Members

3.3.1 Detailed Description

For loop network test.

The documentation for this class was generated from the following file:

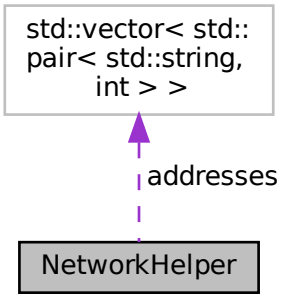
- include/algorithms/loop-network.hpp

3.4 NetworkHelper Class Reference

Control for above transport layer operations.

`#include <network-helper.hpp>`

Collaboration diagram for NetworkHelper:



Public Member Functions

- NetworkHelper (int em_id, const std::string &config_path)
  Construct a new Network Helper object.
- void dump (const char *buf, size_t len)
  Dump the buffer on printing output in formats.
- std::string getLocalIpAddress ()
  Get the local IP address.
- std::string getLocalTime ()
  Get the local time.
- int64_t getFileSize (const std::string &filePath)
  Get the size of a file.
- int recvBuffer (int socketFd, void *buffer, int bufferSize)
  Receives data from a socket into a buffer.
- int sendBuffer (int socketFd, const void *buffer, int bufferSize)
  Sends a buffer of data over a socket.
- int sendFile (int socketFd, const std::string &filePath, const std::streampos read_byte=0, const std::streamsize chunkSize=32768)
  Sends a file over a socket.
- int recvFile (int socketFd, const std::string &filePath, const std::streampos write_byte=0, const int chunkSize=65536)
  Receives a file.
- void send_udp (int target_em_id, const std::string &message)
  Send a message to a process (UDP, original)
- message_t receive_udp ()
  Receive a message from a process (UDP, original)
- int send_tcp (int target_em_id, const std::string &message, const int mode=0)
  Send a message to a process.
- message_t receive_tcp (const std::string &recvfilepath="temp", const int mode=0)
  Receive a message from a process.

Public Attributes

- int **em_id**
- int **procs**
- int **send_fd**
- int **recv_fd**
- std::vector< std::pair< std::string, int > > **addresses**

Private Member Functions

- void setup_send_socket ()
  Setup the sender socket.
- void setup_recv_socket ()
  Setup the receiver socket.

Private Attributes

- std::streamsize **fileSize_toRecv**

3.4.1 Detailed Description

Control for above transport layer operations.

Class responsible for sending and receiving messages / files between processes upon the transport layer, combining with the basic functions of sending and receiving UDP/TCP packets.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 NetworkHelper()

```
NetworkHelper::NetworkHelper (
            int em_id,
            const std::string & config_path )  [inline]
```

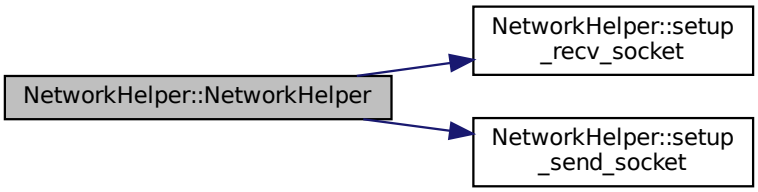
Construct a new Network Helper object.

The constructor reads the configuration file and stores the pairs of IP address and port into vector `addresses`. And then it sets up the sockets for sending and receiving.

Parameters

em_id	The `em_id` of the process.
config_path	The path of the configuration file.

Here is the call graph for this function:



3.4.3 Member Function Documentation

3.4.3.1 dump()

```
void NetworkHelper::dump (
            const char * buf,
            size_t len )  [inline]
```

Dump the buffer on printing output in formats.

This function takes a buffer and its length as input and prints the buffer in hex format.

Parameters

buf	The buffer to be dumped.
len	The length of the buffer.

3.4.3.2 getFileSize()

```
int64_t NetworkHelper::getFileSize (
            const std::string & filePath )  [inline]
```

Get the size of a file.

This function takes a file path as input and returns the size of the file in bytes.

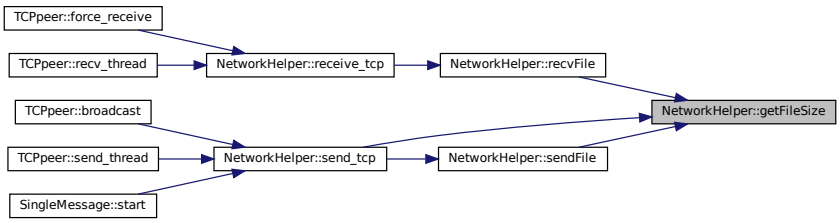
Parameters

filePath	The path of the file.

Returns

The size of the file in bytes.

Here is the caller graph for this function:



3.4.3.3 getLocalIpAddress()

```
std::string NetworkHelper::getLocalIpAddress ( )  [inline]
```

Get the local IP address.

Returns

The local IP address in string format. (e.g. "192.168.0.1")

3.4.3.4 getLocalTime()

```
std::string NetworkHelper::getLocalTime ( )  [inline]
```

Get the local time.

Returns

The local time in preset string format "%H-%M-%S". (e.g. "12-34-56")

Here is the caller graph for this function:

3.4.3.5 receive_tcp()

```
message_t NetworkHelper::receive_tcp (
            const std::string & recvfilepath = "temp",
            const int mode = 0 )  [inline]
```

Receive a message from a process.

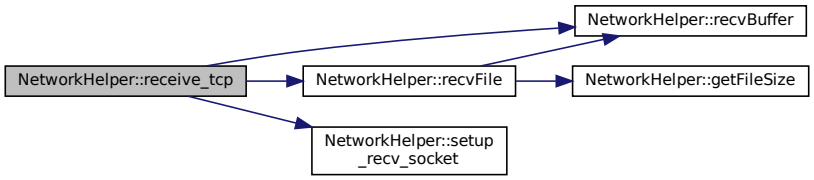
Parameters

recvfilepath	The path of the file to receive, used in mode 1. (default: "temp")
mode	0, receiving buffer and return { `sender_id`, `buffer` }. (default) 1, receiving a file and return { `sender_id`, `recvfilepath` }.

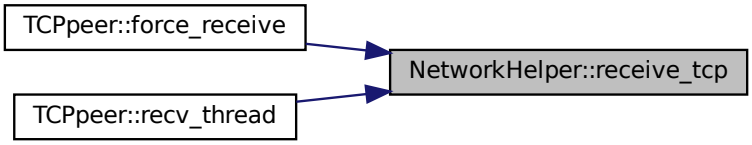
Returns

corresponding `message_t` if success, or {-1, ""} if fail.

Here is the call graph for this function:



Here is the caller graph for this function:



3.4.3.6 receive_udp()

```
message_t NetworkHelper::receive_udp ( )  [inline]
```

Receive a message from a process (UDP, original)

Returns

The received message or empty string if nothing was received.

3.4.3.7 recvBuffer()

```
int NetworkHelper::recvBuffer (
            int socketFd,
            void * buffer,
            int bufferSize )  [inline]
```

Receives data from a socket into a buffer.

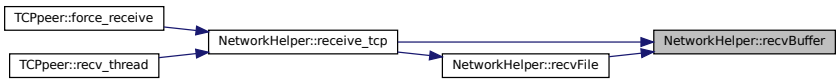
Parameters

socketFd	The file descriptor of the socket.
buffer	The buffer to store the received data.
bufferSize	The size of the buffer.

Returns

The number of bytes received, or -1 if an error occurred.

Here is the caller graph for this function:



3.4.3.8 recvFile()

```
int NetworkHelper::recvFile (
            int socketFd,
            const std::string & filePath,
            const std::streampos write_byte = 0,
            const int chunkSize = 65536 )  [inline]
```

Receives a file.

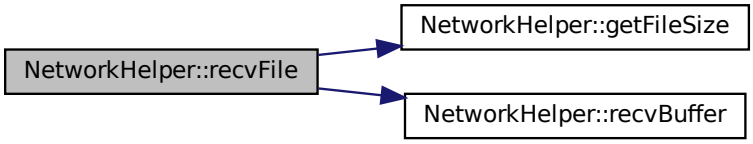
Parameters

socketFd	The file descriptor of the receiver socket.
filePath	The path of the file to receive.
write_byte	The starting position to write to the file. (default: 0)
chunkSize	The size of each chunk to receive. (default: 65536)

Returns

The number of bytes received; or -1 if file couldn't be opened for output, or -3 if couldn't receive file properly.

Here is the call graph for this function:



Here is the caller graph for this function:

3.4.3.9 send_tcp()

```
int NetworkHelper::send_tcp (
            int target_em_id,
            const std::string & message,
            const int mode = 0 )  [inline]
```

Send a message to a process.

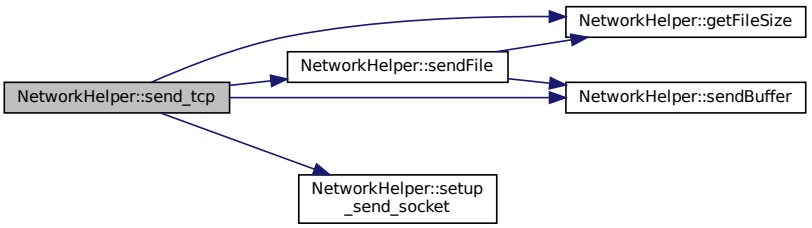
Parameters

target_em↩ _id	The `em_id` of the target process.
message	The message to be sent.
mode	0, sending a string `message` (default). 1, sending a file, `message` field is filepath.

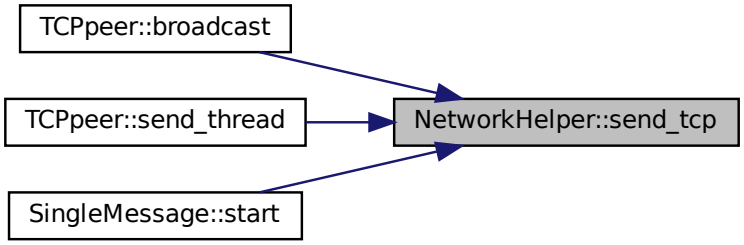
Returns

0 if success or -1 if fail.

Here is the call graph for this function:



Here is the caller graph for this function:



3.4.3.10 send_udp()

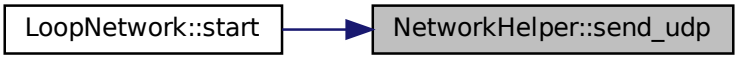
```
void NetworkHelper::send_udp (
            int target_em_id,
            const std::string & message )  [inline]
```

Send a message to a process (UDP, original)

Parameters

target_em↩ _id	The `em_id` of the target process.
message	The message to be sent.

Here is the caller graph for this function:



3.4.3.11 sendBuffer()

```
int NetworkHelper::sendBuffer (
            int socketFd,
            const void * buffer,
            int bufferSize )  [inline]
```

Sends a buffer of data over a socket.

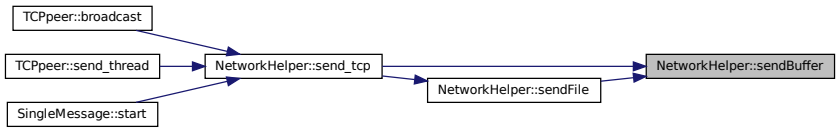
Parameters

socketFd	The file descriptor of the socket.
buffer	The buffer containing the data to be sent.
bufferSize	The size of the buffer.

Returns

The number of bytes sent, or -1 if an error occurred.

Here is the caller graph for this function:



3.4.3.12 sendFile()

```
int NetworkHelper::sendFile (
            int socketFd,
            const std::string & filePath,
            const std::streampos read_byte = 0,
            const std::streamsize chunkSize = 32768 )  [inline]
```

Sends a file over a socket.

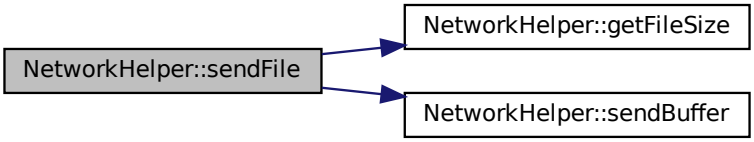
Parameters

socketFd	The file descriptor of the sender socket.
filePath	The path of the file to send.
read_byte	The starting position to read from the file. (default: 0)
chunkSize	The size of each chunk to send. (default: 32768)

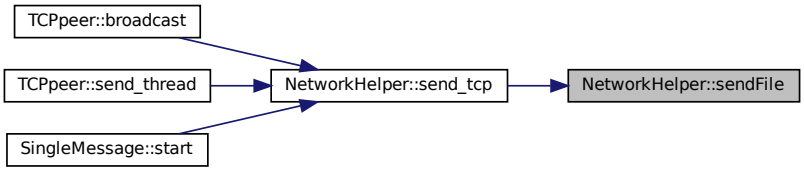
Returns

The number of bytes sent; or -1 if an error occurred before sending any data, or -3 if the file couldn't be sent properly.

Here is the call graph for this function:



Here is the caller graph for this function:



The documentation for this class was generated from the following file:

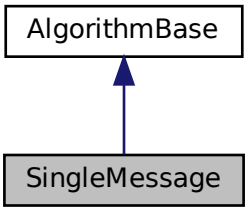
- include/network-helper.hpp

3.5 SingleMessage Class Reference

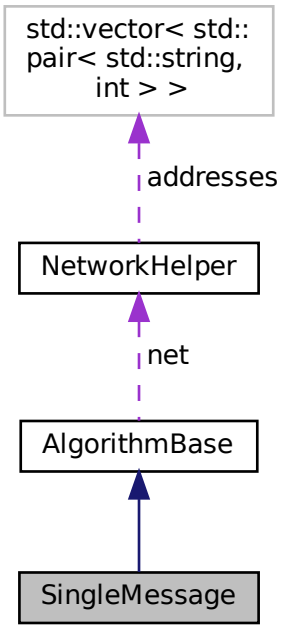
For single message test.

`#include <single-message.hpp>`

Inheritance diagram for SingleMessage:



Collaboration diagram for SingleMessage:



Public Member Functions

- void start (const std::string &message)

  send single message (0->1)
- **AlgorithmBase** (int em_id, NetworkHelper &net)

Additional Inherited Members

3.5.1 Detailed Description

For single message test.

The documentation for this class was generated from the following file:

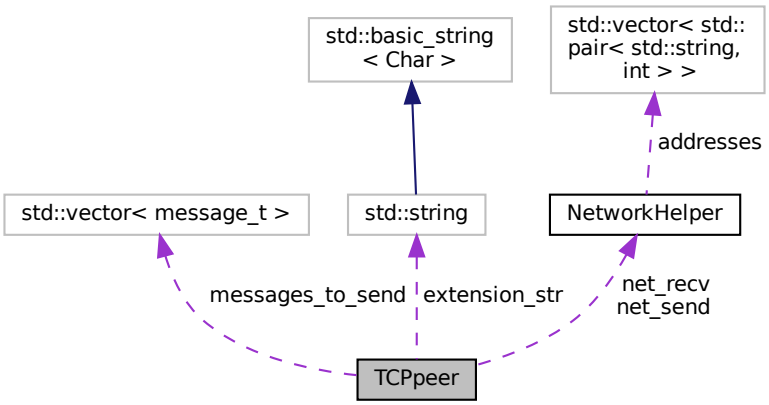
- include/algorithms/single-message.hpp

3.6 TCPpeer Class Reference

Control the TCP peers (send and receive) in two threads.

`#include <tcp-peer.hpp>`

Collaboration diagram for TCPpeer:



Public Member Functions

- TCPpeer (int em_id, NetworkHelper &net_send, NetworkHelper &net_recv)

  Construct a new TCPpeer object.
- void tcp_thread (TCPpeer *obj)

  Create threads for both sender and receiver.
- void broadcast (int em_id, const std::string &message)

  Psuedo broadcast function (send to all other em_id except itself)
- message_t force_receive ()

  Force receive a message.

Public Attributes

- std::vector< message_t > messages_to_send

  Message stack buffer.
- pthread_mutex_t **mutex**
- pthread_cond_t **cond**
- std::string extension_str = ".svg"

  For large file test.

Protected Attributes

- int **em_id**
- NetworkHelper & **net_send**
- NetworkHelper & **net_recv**

Private Member Functions

- void * send_thread (void *arg)

  The send thread function.
- void * recv_thread (void *arg)

  The receive thread function.

Static Private Member Functions

- static void * **send_thread_wrapper** (void *obj)
- static void * **recv_thread_wrapper** (void *obj)

Private Attributes

- pthread_t **sendThread**
- pthread_t **recvThread**
- void * **sendThread_return**
- void * **recvThread_return**

3.6.1 Detailed Description

Control the TCP peers (send and receive) in two threads.

Class responsible for the cooperation bewteen TCP peers. It creates and starts two threads simultaneously, one for sending and one for receiving. The received messages are stored in a stack buffer, and the sending messages are popped from the stack buffer. The threads are synchronized by a mutex and a condition variable.

3.6.2 Constructor & Destructor Documentation

3.6.2.1 TCPpeer()

```
TCPpeer::TCPpeer (
            int em_id,
            NetworkHelper & net_send,
            NetworkHelper & net_recv )  [inline]
```

Construct a new TCPpeer object.

Parameters

em_id	The em_id of this peer
net_send	The network helper for sending
net_recv	The network helper for receiving

3.6.3 Member Function Documentation

3.6.3.1 broadcast()

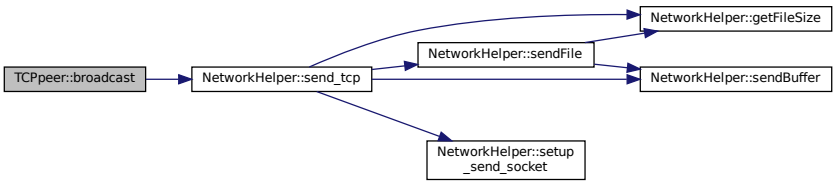
```
void TCPpeer::broadcast (
            int em_id,
            const std::string & message )  [inline]
```

Psuedo broadcast function (send to all other em_id except itself)

Parameters

target_em↩ _id	The target em id
message	The message to send

Here is the call graph for this function:



3.6.3.2 force_receive()

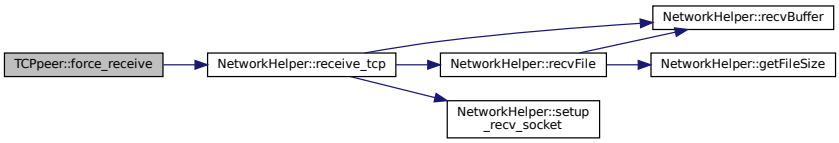
```
message_t TCPpeer::force_receive ( )  [inline]
```

Force receive a message.

Returns

The message received, or -1 if failed.

Here is the call graph for this function:



3.6.3.3 recv_thread()

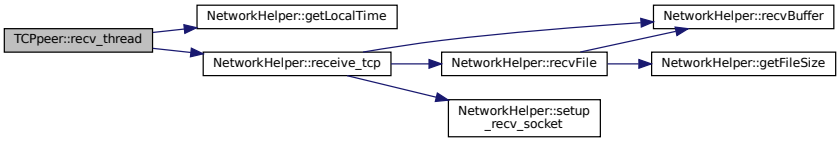
```
void* TCPpeer::recv_thread (
            void * arg )  [inline], [private]
```

The receive thread function.

Parameters

arg	The argument passed to the thread, must instantiated first.

Here is the call graph for this function:



3.6.3.4 send_thread()

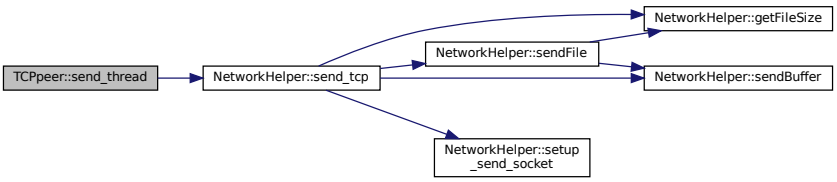
```
void* TCPpeer::send_thread (
            void * arg )  [inline], [private]
```

The send thread function.

Parameters

arg	The argument passed to the thread, must instantiated first.

Here is the call graph for this function:



3.6.3.5 tcp_thread()

```
void TCPpeer::tcp_thread (
            TCPpeer * obj )  [inline]
```

Create threads for both sender and receiver.

Parameters

obj	The TCPpeer object, must initialized first.

The documentation for this class was generated from the following file:

- include/tcp-peer.hpp

Index